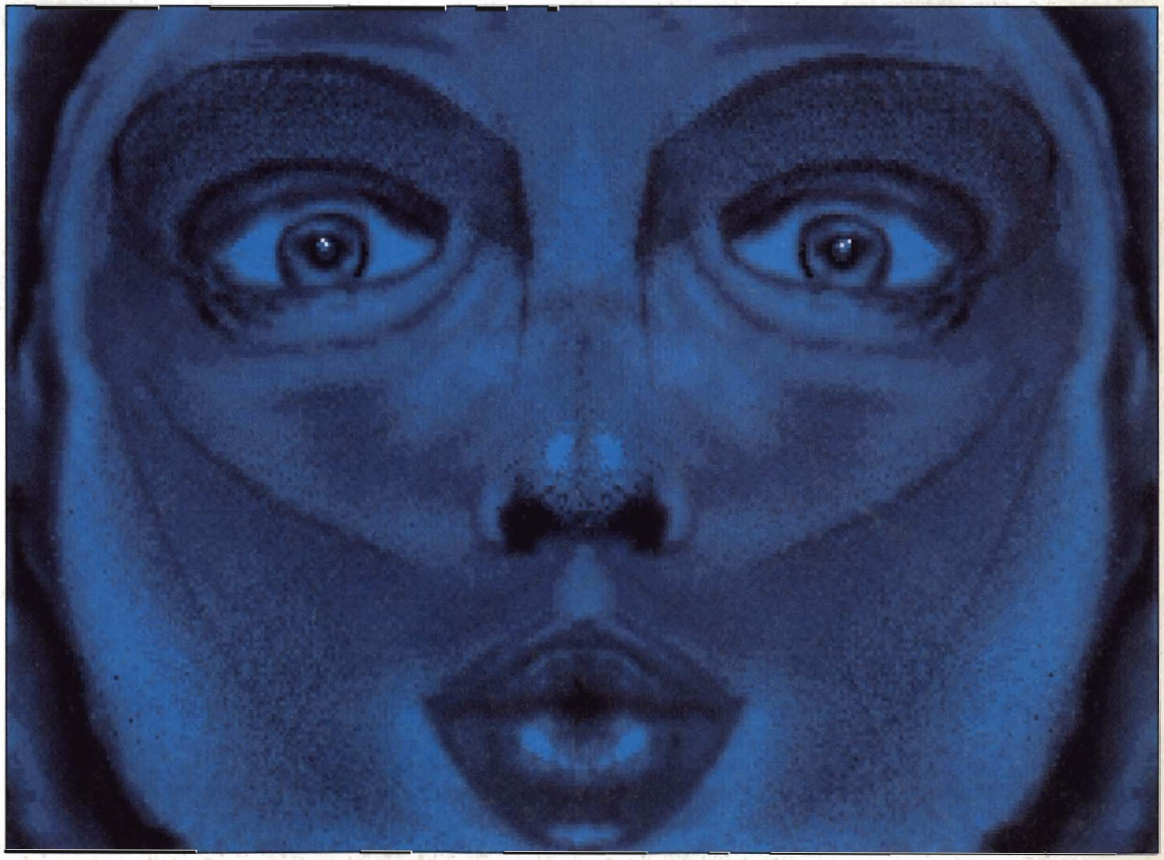
JANUARY 1989: VOLUME 1, ISSUE 4

THE MAGAZINE FOR AMIGA PROGRAMMERS

Canada $4.25
USA $3.50

FOR THE Amiga Transactor

- Calling the CLI from AmigaBASIC *by Jim Butterfield*
- What's changed in Fonts with 1.3 *by Betty Clay*
- The future of Amiga microcomputing *by David Wood*
- Creating run-time libraries *by Steve Simpson*
- Modula-2 programming techniques *by Clark Williams*
- Using TDI's M2AMIGA *by Anthony Bryant*
- Examples of shell programming *by John Faichney*
- Understanding C Declaration Syntax *by Eric Giguère*
- Review of WordPerfect Library *by Jay Gross*
- **Plus** regular columns by Larry Phillips, Don Curtis and Steve Ahlstrom; programming tips in *bits;* new products and more.



Femme Blue by Derek Grime

0 29074 12122 01

EXPLORE THE NEW WORLD

Artificial Intelligence
Expert Systems

You've heard the words. You probably know a bit about what they mean, too. You might associate them with white coats, Russian accents and shiny silver robots. What you might not know is that you can explore AI on your Amiga.

MAGELLAN™ by **Emerald Intelligence** offers performance and interface features UNAVAILABLE on the IBM-PC*, PS/2* or Apple Macintosh*. It was designed, developed and tested on the AMIGA for the AMIGA.

MAGELLAN turns *your* AMIGA into a powerful knowledge engineering workstation. It simplifies some of the complexities of artificial intelligence by using the power of the AMIGA. It is advanced enough to be simple.

The AMIGA's unique capabilities to support inexpensive desktop video, desktop publishing, audio/voice synthesis, extended memory, networking and high-resolution graphics make it ideal for advanced artificial intelligence development. Advanced to the state of being useful, not a graduate-school curiosity. Advanced enough to interface with the real world, and get to work. Advanced enough to be useful.

Use **MAGELLAN** to capture the expertise of your best salesperson, to help junior salespeople "learn the ropes". Build a system to diagnose sick AMIGA's and prescribe repair procedures. Weigh purchasing options in your job or at home. Evaluate performance of the hockey team. Drive animations with rules of behavior of the "actors" or objects. Create original computer art and music. Add real artificial intelligence to games to take them to the edge.

Best of all, use **MAGELLAN** to write programs. No BASIC, no "C", no compiler, no linking, no waiting. No kidding. With programs written with rules, changes can be made interactively — like BASIC — but much, much more efficiently. People don't think in flow charts, or "C" language structures or little curly brackets or parentheses. People think in terms of "IF — THEN". As a child your mother would say "If you take that cookie, then I'll . . ." Rules like this are how people work, how people decide. This is how computers will be programmed by our children.

Welcome artificial intelligence to the real world. Explore **MAGELLAN.**

M A G E L L A N

List Price
$195

Emerald
Intelligence

334 South State Street, Ann Arbor, Michigan 48104. (313) 663-8757.

*IBM-PC, PS/2 are registered trademarks of International Business Machines, Inc. *Apple Macintosh is a registered trademark of Apple Computer Corp. *AMIGA is a registered trademark of Commodore Business Machines, Ltd.

The TOP GUN of AMIGA ASSEMBLERS is:

C.A.P.E.68k!

The "Complete Assembler Programming Environment" from Inovatronics, Inc.!

Count on Inovatronics to get the most speed of development AND execution out of the AMA* assembler standard while putting in more useful control features than any other Amiga assembler. Features like these:

- **Integrated assembler/editor with "assemble from buffer" and "stop on error"**
- **Code format options: EXEcutable OBJect, Binary, ALink format, etc.**
- **AREXX macro support for easy session automation**
- **Program size and number of modules unlimited**
- **Excellent code optimization (optional)**
- **Usage of pre-compiled includes, which net a 10x speed increase over standard includes used by other assemblers.**

Features:	C.A.P.E. 68k	MetaComCo	Devpac	Assempro	Manx
Integrated Editor/Assembler	Yes	No	Yes	Yes	No
Generates Standalone	Yes	No	Yes	Yes	No
Generates BLink-able	Yes	Yes	Yes	No	No
Saves Preferences	Yes	No	No	Yes	No
Optimization	Exclnt	None	None	Poor	Fair

Other features include: complete macro facilities, virtual file handling (for files larger than available memory), a choice of editor command sets (Wordstar or EMACS), and "a rich set of additional directives and pseudo-ops that can be very useful." (Dr. Gerald Hull, Amazing Computing, 6/88)

You will believe an assembler can *FLY!*

INOVATRONICS, Inc.

The REAL POWER in Power Programming.
11311 Stemmons Freeway, Suite 8
Dallas, Texas 75229 214/241-9515

only $89.95!

***Note: AMA refers to the Amiga Macro Assembler, Copyright MetaComCo Plc. Inovatronics products distributed in the U.S.by Southern Technologies, American Software, and Computer Software Services. European Distribution: Burocare Ltd./U.K.Trademarks: Amiga: Commodores Int., Inovatronics and C.A.P.E.68k: Inovatronics, Inc., Devpac:HiSoft, Manx:Manx Software Systems, Inc., Assempro: Abacus, Inc., MetaComCo: MetaComCo Plc.**

Volume 1, Issue 4

Publisher
Antony Jacobson

Editors
Nick Sullivan
Chris Zamara

Assistant Editor
Malcolm O'Brien

Vice-President Operations
Jeannie Lawrence

Vice-President Advertising
Maximillian Jacobson

Customer Services
Renanne Turner

Contributing Writers
Steve Ahlstrom
Scott Ballantyne
Charles B. Blish
Anthony Bryant
Jim Butterfield
Betty Clay
Don Curtis
Matt Dillon
John Faichney
Andy Finkel
Brad Fowles
Eric Giguère
Cathryn Graham
Chris Gray
Charlie Heath
Perry Kivolowitz
Rico Mariani
Louis Markoya
Bryce Nesbitt
Rob Peck
Larry Phillips
Dan Schein
Steve Simpson
John Toebes
Victor A. Wagner
Clark Williams
David Wood

Cover Artist
Derek Grime

for the AMIGA Transactor
The Magazine for Amiga Programmers

The Changing of the Fonts 15
by Betty Clay
How is *your* program going to react when the user selects a multicolour font? Betty looks at what's new in font programming under Workbench 1.3

Calling the CLI from AmigaBASIC 18
by Jim Butterfield
Add power to your AmigaBASIC programs by invoking external commands

The Future of Amiga Computing 26
by David Wood
David looks into his crystal ball to learn about the next generation of Amiga hardware

Creating Run-Time Libraries 30
by Steve Simpson
Cut down your development times and improve program efficiency by building your own shared libraries

Amiga Modula-2 Programming 40
by Clark Williams
Tips on making your Modula-2 programs easier to follow, plus some library housekeeping routines

Programming in M2AMIGA 48
by Anthony Bryant
Continuing this month's special M2 coverage, Anthony shows how to convert C programs to run under this new Modula-2 package, with Rob Peck's 'AudioTools' as a working example

Reading C Declarations 58
by Eric Giguère
Even the most complex of those cryptic C declarations is easily understood if you know how. Never be intimidated by "extern char *const (*foo(char *b))(int, long)" again!

Some examples of shell programming 62
by John Faichney
You can get more done with AmigaDOS scripts than you probably realize. John points the way in this article with some novel twists on a frequently-neglected resource

Departments and Columns

Bits 12

Finding structure sizes
Memory allocation in BASIC
Shorter Manx C programs
Cleaning up on CTRL-C
Finding complete pathnames
Quick'n'easy filled windows
Silent Menus

Amiga Dispatches 4

by Don Curtis

Piracy and copy protection from a law-enforcement point of view; 1.3 release delays; a do-it-yourself BridgeBoard fix; and assorted rumours from the networks

ACCESS - The best of non-commercial software 8

by Steve Ahlstrom

A look at Dave Haynie's *free* disk recovery program *DiskSalv*; new revisions of *Access!*; *CygnusEd* and "other stuff"

ViewPort 10

by Larry Phillips

Can the Amiga still compete? Larry thinks so - and blasts away the arguments of those who don't

Review

WordPerfect Library 73

by Jay A. Gross

WordPerfect's newest Amiga offering isn't perfect, but three out of five isn't bad

About the cover: *Femme Blue* by Derek Grime

Like the cover artists in Volume 1, Issue 1 and Volume 1, Issue 3, Derek Grime also hails from Toronto - it seems there is a lot of Amiga talent in this town. Derek is a graphics artist, and has his own studio where he does a variety of commercial work - including Television ads - using an Amiga.

The picture you see on the cover started out as an experiment to see if "the jaggies" - the jagged appearance of computer-rendered lines - could be avoided by effective use of shading. Incredibly, this picture is in Amiga's low resolution 320 by 200 mode, but by using 32 shades of blue and Deluxe Paint II's shading tools, Derek has managed to give it an extremely non-computer, pastel-drawn look.

TransAmi's
phone number is:
(416) 764-5273

FAX: (416) 764-9262

TOLL-FREE ORDER LINE
1-800-248-2719 Extension 911
(for orders only; have your VISA or Mastercard number ready; available in the U.S. only)

Transactor for the Amiga is published bimonthly by Croftward Publishing Inc., 10-85 West Wilmot Street, Richmond Hill, Ontario, L4B 1K7. ISSN# 0838-0163. Canadian Second Class Mail Registration No. 7690, Gateway-Mississauga, Ont. US Second Class mail permit pending at Buffalo, NY. USPS Postmasters: send address changes to: Transactor, PO Box 338, Station C, Buffalo, NY, 14209.

Croftward Publishing Inc. is in no way connected with Commodore Business Machines Ltd. or Commodore Incorporated. Commodore and Commodore product names are registered trademarks of Commodore Inc.

Subscriptions: see order card at centre of magazine for complete ordering information and prices.

Send all subscriptions to: Transactor Subscriptions Dept. 10-85 West Wilmot Street, Richmond Hill, Ontario, Canada, L4B 1K7, (416) 764-5273. Note: Subscriptions are handled at this address ONLY. Subscriptions sent to our Buffalo address (above) will be forwarded to our Richmond Hill HQ. For best results, use the postage paid card at the centre of the magazine.

Quantity Orders: In Canada: Ingram Software Ltd., 141 Adesso Drive, Concord, Ontario, L4K 2W7, (416) 738-1700. In the USA: IPD (International Periodical Distributors), 11760-B Sorrento Valley Road, San Diego, California, 92121, (619) 481-5928; ask for Dave Bruescher. Quantity orders/enquiries are also welcome from computer/software distributors in the UK, Europe and Scandinavia. Please contact: T.G. Hamilton (W/S) Ltd., Tel: 021-742-5359; Fax: 021-742-2190; or contact Transactor (UK) direct at Unit 2, Langdale Grove, Bingham, Notts. NG13 8SR, telephone 0949-39380.

Editorial contributions are always welcome. Only original, previously unpublished material will be considered. Program listings and articles, including BITS submissions, of more than a few lines, should be provided on disk. Preferred format is Amiga 3.5-inch diskettes with ASCII text files. Manuscripts should be typewritten, double-spaced, with special characters or formats clearly marked. Photos should be glossy black and white prints. Illustrations should be on white paper with black ink only. IFF picture files or Aegis DrawPlus structured graphics files are preferred to hardcopy illustrations when possible.

All material accepted becomes the property of Croftward Publishing Inc., except by special arrangement. All material is copyright by Transactor Publishing Inc. Reproduction in any form without permission is in violation of applicable laws. Write to the Richmond Hill address for a writer's guide.

The opinions expressed in contributed articles are not necessarily those of Croftward Publishing Inc. Although accuracy is a major objective, Croftward Publishing Inc. cannot assume liability for errors in articles or programs. Programs listed in Transactor for the Amiga, and/or appearing on Transactor for the Amiga disks, are copyright by Croftward Publishing Inc. and may not be duplicated or distributed without permission.

Production
In-house with Amiga 2000 and Professional Page

Final output by Vellum Print & Graphic Services, Inc., Toronto

Printing
Printed in Canada by Bowne of Canada Inc.

Amiga Dispatches

Copy protection and a home BridgeBoard repair

by Don Curtis

Don Curtis is a police officer in Denver, Colorado, who has for the last two years been assigned to program design and development along with system design and maintenance of ten AT&T Unix™ 3B2 computers. In his spare time, Don is an assistant sysop on CompuServe's AmigaForum.

There are a few hot topics that have been running around the networks of late. It seems the hottest issues are copy protection and the if/when of v1.3 of the Amiga OS.

Before I get into this discussion of copy protection, let me make one thing very clear. *Not everyone is a software thief!* There are thousands and thousands of honest folks who buy their software. They do not steal it. This issue is not unique to the Amiga or any other computer. Software has been stolen for mainframes, minis, micros and tinker toys. No segment of the computer industry is immune to software theft and no segment consists entirely of software thieves. Just like everything else in life, there are those who are 100% honest, 100% dishonest and every place in between. This discussion is not directed at any one person, publisher, group or manufacturer. It is general in nature and please read it as such.

For the purposes of this discussion, I'm going to lump 'use protection' with actual disk copy protection. Use protection is when you require some external device, such as a dongle or manual keyword to use the software, but the disk itself can be copied like any other disk.

I have a somewhat unusual perspective when it comes to copy protection. I've co-authored some commercial software, I'm a software purchaser and user, I've dealt with software publishers and, by profession, I'm a police officer. That gives me the ability to see all three sides of the coin. *Three* sides, you say? Sure: there is the viewpoint of the publisher and author, the viewpoint of the end user, and the viewpoint of the law enforcement community when software theft is reported to them.

The publisher and author don't always agree on the issue of whether the author's program should be copy protected. I've seen lots of cases where the author wanted the software published with no copy protection but the publisher insisted on it. However, that aside, both the publisher and author agree that they'd like to make a fair and reasonable profit for their work. The software purchaser would like some assurance that the copy protection won't prevent them from using their program in an easy and convenient manner. The law enforcement community would like understandable laws and reasonable methods of enforcement so that they can take action against software thieves. They would also like a bit of training so they know what they are doing.

All put together... the above opens up several cans of worms. The publishers and the users are at odds with each other and the law enforcement folks are stuck in the middle. Those publishers who copy-protect their software have honest users upset with them. Those publishers who don't copy-protect get upset with the dishonest users who give away their software and, for the most part, the police haven't the faintest idea what is going on.

With this in mind, let's go back to a few base issues. Software, by its very nature, can be copied over and over again, without any degradation. Unlike photocopies, video tape copies, audio cassette copies and the like, when you make a copy of a software program, it is every bit as good as the original. No noise, 'weak bits', loss of resolution or anything like that occurs. Such is the nature of digital material. No matter how many generations (copy of a copy), each is indistinguishable from the original. This, of course, means that a software thief can make a copy of some material, post it on a BBS, and the software can be illegally distributed world-wide in a matter of hours, with each copy being a perfect replica. As a side note, that's exactly why the record publishers are so upset about the new Digital Audio Tape (DAT) machines.

With printed material, video and/or audio material, if something similar were tried (and it is), each generation is worse than its 'parent'. When you get to about the fifth or sixth generation, the material is almost useless. That means that to steal the material, you need a second or third generation tape to duplicate if the thief expects to get something worthwhile.

While there is a significant amount of theft in the video and audio markets, the need to get a quality 'original' to duplicate slows things down considerably. To make 10,000 copies of a video tape (without expensive duplicating equipment) would take months. With software, 10,000 copies can be made in less than a day if the illegal networks all clicked along at top speed. And, no special equipment is needed - just a computer and a standard disk drive.

No wonder the publishers are worried. One person can give a copy of the software to ten friends, who in turn can give it to ten more friends, and so on down the line. The few minutes required to duplicate the disk is no big deal. There are programs out there that can take an entire disk and compress it into a single file that can be posted on an illegal BBS and obtained via a modem.

Does copy protection even work? There is no real answer to that particular question since, no matter what is done, someone, somewhere, will break the protection and give away the software. Trip Hawkins of Electronic Arts made an interesting observation. He stated that, when EA started to offer DeluxePaint in unprotected form, sales of DeluxePaint suddenly dropped by 50%. There is no way to tell with a single figure like that if the drop off was the normal drop in software sales after the first few months or if it was due to a sudden increase of illegal distribution. But it is an interesting observation.

If the drop was due to a sudden increase in software theft, then the original copy protection (as weak as it was) did work in that it prevented casual theft. That means that Joe Dishonest User couldn't just make a quick DiskCopy and give DeluxePaint to his best buddy. If the drop was the normal drop experienced in software sales after the first few months, then the copy protection didn't work, so why have it?

One of the next issues is: if you are going to copy-protect software, which software should be protected? All of it, games, expensive software, easily used (no manual needed) software or some other category? Many folks have expressed the opinion that it's OK to place copy protection on games but that serious software should be sold unprotected. These folks state that, because the life of a game is only about 90 days, and because games are probably the most often stolen pieces of software, the publishers need every edge they can get. I happen to agree with that particular point, because games are particularly susceptible to casual copying. Productivity software, on the other hand, has a much longer selling life. It should be installable on a hard drive, and the user should be able to use the software with no worry about losing a dongle, or manual keywords, or the copy protection failing. The failure of a program because something went wrong with the copy protection could cause a business to lose sales. Meanwhile, the lack of copy protection could cause the publisher and author to lose sales. Which is the lesser evil? Does it really matter if the publisher and author deal with game software or productivity software? The loss of money, either by publisher and author or by the user, is wrong no matter how you look at it.

Another often talked about subject is: what kind of protection should there be, if there is going to be copy protection at all? The use of disk-based copy protection has a few problems. First, the user has a hard time making an archival backup of his software. Making archival backups is perfectly legal and also a good idea. Programs like Maurader II or Mirror can help, but that's added cost to the end user. There have been complaints that certain types of disk-based copy protection cause damage to disk drives. That, of course, is totally unacceptable. Disk-based protection also makes it difficult or impossible to install the program on a hard disk. Some programs require a 'key' disk, allowing you to put your runnable copy on a hard disk, but requiring you to insert the key disk before the software will run. For productivity software, the inability to use it on a hard disk can, and should be the kiss of death. Even worse, having a key disk go bad at a critical point can cause the user no end of headaches and possible lost business.

Non-disk-based protection would seem to prevent the above. You can copy the disks freely, so making archival copies is no problem. Unfortunately, in the design of the Amiga, the lines to the game ports were left unbuffered. This means that, under certain conditions, a hardware device such as a dongle can zap your machine. A manual keyword doesn't have that problem but, since a manual can be photocopied, neither does it offer real protection.

Many programs in the MS-DOS arena use a dongle attached to the parallel port. They are completely transparent to the user, except when the user happens to have several programs that use the same type of protection. Then the user has to pile one dongle on top of another. I've heard of a situation where the user had a string of dongles hanging off the back of the machine that was about 15 inches long! They apparently work okay together, and I suppose you could make a short cable to hook from the port itself to the dongle so that they wouldn't have to stick straight out behind the machine, but it really is awkward. All the same, they do seem to work. I'm certainly not an expert on what software is broken and which isn't, but so far I've not heard of this type of protection being broken.

One other popular form of non-disk based protection is the "keyword from the manual" technique. Before the program will run, you must look in the manual, on a certain page, line and word count, and enter that word before the program will run. That is probably the least harmful of all the protection methods, and also probably one of the least effective.

There have been suggestions for different kinds of non-disk-based protection: things like the use of special data encryption chips and unique user identification numbers... Fingerprinting software and forcing the purchaser to register with the manufacturer... Hidden serial numbers, embedding Visa credit card numbers in the software, self-modifying software and lots more. The general idea behind all of them is to identify the original purchaser somehow so that, when a stolen copy is found, the person who originally bought the software can be prosecuted for distributing illegal copies.

Here's where my law enforcement background comes in. If I buy a copy of SuperGame II, how does the publisher identify positively who I am? Must I produce three forms of ID before I can buy the product? Must the dealer take a set of fingerprints before I can buy the product? What if I give a phony name when I order directly from the manufacturer? In other words, there really is no way to ensure that 'Joe User' really is Joe User. Even if, somehow or other, the publisher could positively identify Joe User as the person who purchased the software package with serial #a12345, how does the manufacturer (or the police) prove that Joe was the person who illegally distributed the software with the same serial number on it? What if someone broke a different copy and changed the serial number to #a12345? What if Joe's 'friend' came to Joe's house, made a copy and distributed it without Joe's knowledge? Any number of possibilities exist where Joe may not have distributed the software. To prosecute, you must be able to prove beyond a reasonable doubt not only who Joe is, but that he also was the person who distributed the software illegally.

In other words, there is no way you can prosecute Joe. So here we go again: if it doesn't do any good, if you can't prosecute the original owner, why do it at all?

So what is the answer? How do the publishers insure themselves against dishonest users, and how do honest users get quality software with no copy protection hassles? That, my friend, is the three billion dollar question. According to Rainbow Technologies, the manufacturer of one of those parallel port dongles for MS-DOS machines, 3.5 billion dollars of microcomputer software will be sold in 1988 (world-wide), and another three billion dollars worth will be stolen. If I knew the answer to that question, I'd be a rich man today... and I'm not. I do know that, of that three billion dollars worth of stolen software floating around out there, not all of it would have been bought if everyone was totally honest. Much stolen software gets used once and put on a shelf, never to be used again. The thief would have never bought the software in the first place, so the publisher really hasn't lost a sale. This does *not* make that form of software theft any less palatable. It is still completely wrong. I would suggest to the publishers that they try to understand the user's point of view - that they don't copy-protect productivity software or, if they must, that they use non-disk-based protection schemes. I'd like them to provide demos that would allow folks to try the software before they bought it. The demo software could be crippled in some manner (such as not allowing it to save a file) so folks couldn't just use the demo rather than buying the real thing.

I would suggest to users that you work with the publishers in an effort to stop software theft. Let them know what type of protection is acceptable to you and support them when they try to meet your wishes. Work through local users' groups, trying to educate them to the perils of software theft. Get them in turn, to try to educate their friends who also own computers (no matter what brand). Try to get decent laws passed that the police can understand and enforce. Work with the law enforcement people to try to get illegal BBSs shut down and their equipment confiscated. Have your local newspaper run articles educating the general public to the problem of software theft.

Do whatever you can to lower software theft (you'll never stop it completely - that's not a realistic goal). Maybe, just maybe, our efforts will be recognized by the publishers and they'll see that they can publish a program completely free of protection without worry.

Amiga OS v1.3

As I write this, it's the beginning of September *[Sorry, folks! - Ed.]*, and v1.3 still hasn't been released. So what the heck happened to it? Well, the way I understand it, more problems were found in the Fast File System (FFS). CBM refuses to release any of 1.3 (other than the early printer drivers and new preferences) until it's all complete and working properly. I agree with CBM in this case, but boy does it have some folks hot under the collar? Since the two major things in v1.3 are autobooting and FFS, it only makes sense for these two to be solid before release of the software. Can you imagine the users' reactions if CBM released FFS with bugs in it, and suddenly they lost several days of work off their Hard Disk? I'd much rather hear screaming about the slow release of an Operating System upgrade than about a buggy Operating System. In any event, supposedly knowledgeable folks are now saying that v1.3 will be released on September 29th. By the time you read this, you'll know if that's true or not. By the way, that is not an 'official CBM release date'; the last *official* release date I had was, 'when it's ready.'

A BridgeBoard fix

Having problems with your BridgeBoard? Was it working and suddenly become flaky? I've heard from quite a few folks about similar problems, and they cropped up for me yesterday. The BridgeBoard had been working perfectly up until yesterday, when it suddenly failed during the BindDrivers at boot-up. The familiar 'gronk' of my two 5.25" drives didn't occur and the boot hung. That brought on the familiar 'Software Failure' requester and subsequent guru.

I had just moved my computer desk and, although I had been very careful not to jar anything, I'm also aware that the Bridge is very sensitive to proper installation in it's slot. Hoping that was the problem, I opened the case, lifted and pushed down on the Bridge to make sure it was properly seated. Re-boot and check it out. Yup, the Bridge booted and worked just fine. "Great!," I thought, until I did a warm start about an hour later. Darn thing failed again. Well, not to be outdone by a machine, I pulled and pushed again. I powered on and the drives 'clacked' instead of 'gronked', but the boot continued apparently normally. When I went to use the MS-DOS side, though, it wouldn't recognize my hard drive or an MS-DOS boot disk and, in fact, it wouldn't recognize keystrokes either.

Thinking that perhaps an intermittent solder joint or dirty contacts may have been the problem, this time I removed the

BridgeBoard and cleaned the contacts. I then inspected as many joints as I could and found a few I thought were suspicious. I reheated those joints and added a bit of fresh solder. (By the way, if you don't have the proper equipment and/or don't know what you are doing... do *not* try to solder on the BridgeBoard. It is a four-layer board and you can easily ruin it with improper soldering techniques!) I re-installed the board and booted. Well, the drives gronked this time and the boot seemed normal; but, again, it wouldn't recognize the hard drive or the disk drives. What the heck could be wrong?

I tried removing and reinstalling both the BIOS ROM and the floppy disk controller (FDC) chips. Now, all I got on boot up was the guru again. Well, I have two other cards in there, a multi-function/RAM expansion card and the hard disk controller. I figured the multi-function/RAM card was fine since, when I was able to get to the PCWindow, I could see that it had found and tested the extra RAM. That meant that the basic operation of the Bridge seemed okay, but drive access was screwy. Drive access? Of course! Pull and push on the hard drive controller, boot up and everything was fine! I tested it for a while and it worked like a champ, just like it always had. Apparently the hard disk controller had gotten slightly out of line and was causing all my problems.

Keep that in mind the next time you have a problem with the BridgeBoard. If it suddenly stops working for you, or it's been acting flaky, check to make sure that the BridgeBoard is seated properly. Make sure the chips aren't loose in their sockets and also check that any other cards on the Bridge side are also seated properly. You may save yourself an unnecessary trip to the repair shop (and the associated bill).

Rumours from the networks

A few other things that are floating around the systems are that CBM should be releasing a hard disk for the A500 about the time you read this. Version 1.4 of the operating system (with major upgrades to Workbench) is slated to start testing about now and, if everything went like clockwork, could be ready as soon as the early part of next year (late January?). While v1.3 will not require new ROMs (unless you want AutoBoot), version 1.4 *will* require new ROMs to be usable. Those of you with A1000s will only require a new KickStart disk (for v1.3 or v1.4), which will be included with the upgrade.

I keep getting questions about the A2500 machines. The main concern expressed in those questions is whether the A2500 series is a new machine or not. If you don't already know: no, the A2500 series is nothing but an A2000 with some new boards and/or other goodies. The extras will be available separately. Anyone with an A2000 (B2000 motherboard) can upgrade it. In that vein, I've also been informed that the A2500AT will never see the light of day as an 'off the shelf item', but the AT-style BridgeBoard should become available this year. The A2500UX UNIX™ machine should be available as an off the shelf item or as separate components. Reported release date is also this quarter.

The fastest
Modula-2
Software Development System for
AMIGA
$ 249
Demonstration disk $ 10

Modula-2 is the successor of Pascal. Both languages were invented by Professor Niklaus Wirth. Modula-2 is an all purpose language suited for educational, home and professional use. Modula-2 leads to good programming technique, easy readability and maintainable programs. In fact, a lot of schools and universities have chosen Modula-2 for their programming courses.
M2Amiga is a software development system for Modula-2 based on an extremly fast single pass compiler. It is fully integrated into Amiga Workbench and is easy to use from CLI or Workbench. It runs on all Amiga models with a minimum configuration of 512k RAM and one disk drive.
M2Amiga has been developed specially for the Amiga and therefore optimally supports the features of this unique computer.
- Single pass compiler produces optimized native MC68000 code from standard ASCII files. Full support of latest edition of Modula-2.
- Fastest library calls possible on the Amiga. No intermediate code needed for library calls and parameter passing.
- Supports FFP, 32 bit IEEE and 64 bit IEEE reals formats, you can even mix all three formats within the same program.
- Easy access to MC68000 registers, allows the insertion of inline code.
- Full interface definition to the Amiga's operating system following the official definitions.
- The best runtime system currently available on the Amiga. Opens and closes all used libraries, captures all GURUs and releases all used resources after program crash or termination. Routines for System requesters enable user interface consistent with the Amiga's Intuition.
- The emacs based editor places the cursor on compilation errors and explains them in full text.
- In just a few seconds the linker produces stand-alone applications that can be run from CLI or Workbench. Stand-alone applications do not contain any useless overhead – they are compact and fast.
- All programs can produce icons for their output files to allow effective use of the intuitive Workbench environment. Experience the new feeling of program development by selecting and double clicking icons!
- Also included are a project management tool and other utilities.
- Besides the handy manual, an introductory guide to Modula-2 is included. This documentation allows both the beginner and the advanced user to find any information needed quickly and easily.
- Demo programs show the use of various features of the Amiga.
- For professional program development special utilities are available.
  + Source Level Debugger, the new way of debugging your programs. Inspect all your variables (displayed in their own format), zoom into structures as records or arrays or follow pointer chains by clicking the mouse buttons
  + Object file converter, converts standard Amiga object files into M2Amiga format. This allows linking of routines written in other languages.
  + Linker for devices or libraries. Add power to the Amiga by supplying new libraries and devices. Easy and intuitive programming instead of fooling around with Assembler and Amiga linker are possible now!
  + Modula-2 Amiga Programming System Environment

We have Modula-2 Compilers for HP-UX, IBM/370, PCs (Taylor, M2SDS, JPI), OS-9 and Sun, tomorrow even more!
Texas residents add 8% sales tax. International Orders add $ 20 shipping.

The Modula-2 people:
INTERFACE TECHNOLOGIES
3336 Richmond, Suite 323, Houston, TX 77098-9990 (713) 523 8422
Dealer inquiries welcome

2U
A. + L. Meier-Vogt
Im Späten 23
CH-8906 Bonstetten/ZH
Switzerland
Tel. (41) (1) 700 30 37

ACCESS

The world of non-commercial software

by Steve Ahlstrom

As the primary Sysop on CompuServe's Amiga forums, and a veteran Amiga programmer, Steve Ahlstrom sees just about every notable public domain and shareware program that gets written, and in this column he tells you about the ones that have recently caught his eye. If you have some information you'd like to share through this column, you can leave a message to Steve on CompuServe (his PPN is 76703,2006), or send the info to us and we'll pass it on.

DiskSalv: shareware from David Haynie
No fee requested

Have you ever put a disk in and had a system requester come up telling you the disk can't be validated? Or that it's not a DOS disk? Or maybe your worst fear has come true - your hard disk has just developed a 'key' error and it can't validate? And of course, Murphy being the man who gets to us all, you don't have a current back-up. It's panic time!

CBM gives us a little utility called DiskDoctor. Unfortunately, it's not worth much. DiskDoctor tries to save the day, but it has the problem of writing to your original disk - you know, the one you're trying to resurrect. While DiskDoctor usually works, the fact that it writes to your original disk means that there are times when it will make matters much worse.

DiskSalv can save the day. This is a program written by Dave (Hazy) Haynie, a design engineer at CBM (if you own an A2000, take a look at the names on the motherboard, Mr. Haynie is right there). It is the program that should have been included in your c: directory instead of DiskDoctor.

DiskSalv (v1.3) is a disk recovery program for all Amiga file system devices that use AmigaDOS 1.2 or 1.3. It fully supports both the Old File System (OFS) and the new 1.3 Fast File System (FFS). DiskSalv will scan a bad disk volume for anything that can be recovered and will restore the data it finds to any AmigaDOS volume. It does not write to your original disk.

The fact that you can restore to *any* volume is important. It means that, given enough memory, you can restore to RAM: or other RAM-based devices. Since the original disk is only read from, should there be a problem that DiskSalv cannot recover, your original disk is still intact so you might try other methods of recovering the data.

DiskSalv works fine on hard disks. When you run DiskSalv to recover your hard disk, you will more than likely specify the output to go to floppy (you don't have *that* much RAM, do you?). If you run DiskSalv with the FORMAT command line parameter. This forces the output device to be formatted before any output files are written to it.

DiskSalv is run from the CLI. In its simplest form, without special command line options, it is used very much like the AmigaDOS DiskCopy program. To recover bad files from DF0:, restoring them to DF1:, you would type:

1> DiskSalv [FROM] DF0: [TO] DF1:

The FROM and TO keywords are optional as long you enter the *input* device name first. DiskSalv will then display:

```
DiskSalv V1.3 Copyright 1988 by Dave Haynie
Salvage FROM Device DF0: TO Device DF1:
DEVICE     =   trackdisk.device   (DF0:)
UNIT       =        0   FLAGS     =       0
HEADS      =        2   SECTORS   =      11
LOCYL      =        0   HICYL     =      79
LOBLOCK    =        0   HIBLOCK   =    1759
RESERVED   =        2   MEMTYPE   =       3
ROOT BLOCK =      880   DISK SIZE =    1760
Scan Range: START 2, STOP 1759, Standard FileSystem Expected
Should I continue [Y]?
```

DiskSalv goes through many steps to salvage your disk. Initially, the program will scan your disk looking for valid AmigaDOS file or directory blocks. The program will open up a small window with three headings: BLOCK, NODES and TYPE.

The numbers under the BLOCK heading will fly by pretty quickly. This is just an indication of where in the process the program currently is. If you're salvaging a floppy, the block number will range from 2 to 1759. Under the TYPE heading

will be listed the type of block at which the program is currently looking. It may be a file block, root directory block, data block, unused block, unknown type, or a bad block. If you are salvaging your hard disk and you are using the Fast File System, there is no way to tell the difference between data blocks and free blocks during this scan. This is due to changes made from the original file system to the new file system. These blocks will be shown as ???? (unknown). Under the NODES heading will be the number of directory and file blocks that have been found so far.

After doing a lot of detail work, building a directory map, linking files with their associated directories, killing off empty directories, and other technicalities, the disk structure is restored to the output device. This restoration will continue until a CTRL-C abort, or the output device is full, or the operation is complete. If the output device is full and there is more to be done (as in restoring your hard disk to floppy) DiskSalv allows you to insert a new disk and then continue.

DiskSalv uses normal AmigaDOS I/O routines to recreate the recovered files. It can restore to a subdirectory instead of the root of a device. If a subdirectory is specified that doesn't exist, DiskSalv will create it for you. It gracefully handles file name collisions, which occur when a file already existing on the output device has the same name as one being restored. In this case, the colliding file is renamed, with an extension ranging from "-0" to "-99".

DiskSalv has saved me innumerable times and I'm sure you'll find it a very necessary tool to have in your arsenal of utilities.

Mr. Haynie does not request remuneration for DiskSalv. To quote from his documentation:

This program may be distributed free of charge, provided that no extra restrictions are placed on it. Nominal charges for copying or on-line services are permitted provided that they are only for those services. This program was written to help out the Amiga community, not to make folks feel guilty. Thus, no payment is required for its use...

If you really want to send money, consider sending a donation instead to:

GreenPeace
1436 U Street NW
Washington, DC 20009

Access! update

In my last column I talked about Access!, the shareware terminal program by Keith Young. That column was written in June. The prolific Mr. Young has turned out several revisions in the intervening months. The version I talked about was version 1.31. He is now (as I write this in November) up to version 1.42. Many enhancements have been made, while the overall program size has actually decreased.

Font support has been added (non-proportional fonts only). It's now a menu item. You have the choice of selecting what font you want to use in the main window and in the chat window. Unfortunately, in 1.42, you cannot save your font selection (that feature is promised for the next version, which will also offer full ARexx support and is due 'RSN'). A program called RD has been added. It is a file reader similar to More, Less, etc. It is very well done and is fast. Put it in your c: directory and you can use it from Access! or as a stand alone program.

Other changes include the addition of full scripting and, for users of William Hawes' WShell, the ability to open a CLI window on Access!'s own screen. Even if you are already an Access! user, be sure to look at the latest version. In my opinion, Access! is the premier Amiga communications program available, whether commercial, shareware, or public domain.

CygnusEd

One of the nicer shareware text editors has gone commercial. CygnusEd has evolved into CygnusEd Professional and is being marketed by ASDG Inc. A large number of people swore by the shareware version of CED. It has got a lot more powerful and features a full ARexx interface. While I was not one of the 'faithful' as far as the shareware version goes, I picked up a copy of CEDPro and said goodbye to the text editors I'd been using before. This editor is *hot*! If you are a registered user of the shareware release, you may upgrade to the commercial one by sending $25 to ASDG. Suggested retail of CEDPro is $99.95.

Other stuff

There are strong indications that a couple of major electronic CAD packages will shortly be availble in the shareware marketplace. More on this as the situation develops.

While not in the utilities or applications genre, there are a lot of 'fun' programs and demos out there. Hopefully by the time you read this, both Dragon's Lair from ReadySoft and Dungeon Master by FTL will be on your dealer's shelves. If not, *get the demos*. Incredible stuff!

Louis Markoya has released a fantastic animation called Whale. It was created with Turbo Silver and Terrain. It features a relaxing, undulating ocean. As you watch, you can almost feel your blood pressure lower. It's beautiful to the eyes and soul. Then, when you least expect it, a whale comes up from the depths and arcs out of the water in front of you. All that's missing to really give this animation that finished touch is sound. The song of the whale as he makes his dive would be the coup de grace.

Keep checking your local bulletin boards and the commercial services for some real treats. Amiga people are very prolific and chances are you'll find that needed utility, tool, game, or animation that is exactly what you need. If you come across something you think the world should know about, please send it along!

ViewPort

Does the Amiga still compete?

by Larry Phillips

Larry Phillips is an Amiga hardware and software hacker from Vancouver, and a Sysop on Compuserve's Amiga forums.

Every time a new machine is announced, there appears a rash of cries from Amiga owners. "The Amiga is falling behind!", "We aren't as fast as the [fill in current announcement here]!". The cries are of doom and gloom, lamenting the lack of speed, software, graphic capability, sound capability, or peripherals. Listening to these complaints, one would think that if the next announcement from Commodore does not match or beat the power of a Cray, that the Amiga will be immediately condemned to take its place in history with the hoop skirt and the Coleco Adam.

Does this constitute fair comment on the state of the Amiga? I don't think so, when you take into account not only capability, but price as well. The last few major machine announcements have been pretty 'ho-hum' from where I sit.

IBM's PS/2 was, as the popular expression stated, "Yesterday's hardware, Today.", and its software was (and still is), "Yesterday's software, Real Soon Now." It made no difference to the doomsayers though, the promise of future multitasking and 256 colours on-screen was enough to wind them up and get them crying into the networks about how CBM was falling behind.

Never mind that the low end PS/2 is 'separate' from the rest of the line in capability, not having nearly the speed, power, or features of its more able brothers; never mind that to get more or faster processing power than that of the Amiga, one needs to buy a PS/2 equipped with an 80386 CPU chip with its mortgage-sized price; never mind that even then, you still do not have the graphic or sound capabilities of the Amiga, the out-of-the box multitasking capabilities of the Amiga, and were, until just recently, saddled with some absolutely ludicrous and arbitrary limitations on things like the size of hard drive supported. Let me know when they finally get multitasking right, in somewhat less than two Megabytes of memory, and then I'll take a look and decide if the Amiga is falling behind.

No sooner had the dust settled than the Mac II arrived on the scene. Again the hue and cry resounded through the Amiga community. Faster, more colours, more software, all combined to whip the crowd into yet another round of flailing at CBM, wanting them to produce a 68030 machine with oodles of memory, or Unix, or more colours available. Sure, the Mac II is faster. The 68030 is a faster chip than the 68000, but again, one look at the price, or more importantly, at what things the Amiga community purports to hold near and dear to their hearts, will show you that the complaints are unwarranted. More colours in total is definitely nice, but are you willing to pay the price for that feature? The Amiga is no worse off now than before the Mac II introduction. We can still put all 4096 colours on the screen. Ever see an animation on a Mac II that is as smooth and as fast as on an Amiga? We can still run multiple programs at once without having to have them especially written to do so, in the manner that Apple laughably calls multitasking. AUX (Apple's Unix-alike) is a pretty good thing, but at what price? The resources required to run it are not cheap. You can multitask an Amiga right out of the box, and that has not changed.

Now we get to NeXT, the "Mac II - alike" from Steve Jobs. Heck, the beast isn't even released yet, has not had a retail price set yet, and already the whining is showing up on the networks. Take a good, close look at the NeXT machine. It has a 68030 that should run much faster than it does, some nice software, and a very good sound chip that will double as a means of interfacing with the real world in a number of ways.

In my opinion, the only real revolutionary step is the bundling of the optical disk drive with the system, and even this is a

questionable move. It remains to be seen whether it will be accepted, or whether virtually 100% of them will be sold with an additional hard drive for speed. Will the users demand a floppy drive for the purpose of loading up application packages? Will any manufacturers release applications on $50 media for the optical drive? Will the machine ever have the ability to change optical media without having to reboot? Will the machine be a giant, black paperweight without a full network connection? Last, what will the retail price be? NeXT is now specifically targeted toward educational institutions, so much so that the only price announced is an educational price. Current speculation is that the final, retail price will be $10,000 or more.

My general reaction to the complaints I hear is to ask the complainer what he expects for 1/3 or less the price of these machines, and what he thinks of the things the Amiga does right now, let alone at some future date. We have full multitasking with no fuss or muss and no special provision for it in program code. We can do things that IBMers have been trying to do for years with their TSR (Terminate and Stay Resident) programs. We have fast graphics, more colours on screen, quite good sound, hordes of application software (much of it low cost or public domain), and all in all, a very capable computer for its price. You can do things with an Amiga that the other machines can only hint at. That we cannot do everything better than every other machine is no cause for panic.

The big question is "Are we really falling behind?" The answer to this depends on what you are looking at. Yes, there are faster machines, machines with more colour, better sound, larger disk storage capacity, etc. There is a price to be paid for these features though, in both money and in reduced capability in other areas. No other company has yet announced a machine that is better in all ways than the Amiga, in any affordable price range, and I don't call that falling behind.

If you look at the recent developments from IBM, Apple, and NeXT, you can see the effect that the Amiga is having upon the industry. They have finally decided that multitasking is a good thing, that lots of colours is useful, and that people really do want sound. I can't even begin to tell you how much I chuckled when IBM increased the size of hard disk supported to 2 Gigabytes.

Throughout all these announcements runs a single, common thread that is obvious to me. The Amiga is ahead, and they are worried that one day the whole computer community will wake up and demand more from their machines. Want to bet at the number of businessemen that now use VGA in their everyday operation? After all, IBM has "blessed" it now, so it must be alright to use it.

Now the A2500 in its various forms has been announced. A 68020 machine running the processor at double the speed and having more memory and a built in hard disk; you would think that it would alleviate some of the complaining, but alas, it is not to be. Now the complaints focus not on what the another manufacturer's machine can do, but rather on "what the A2500 could have been". They seem to want Commodore to provide them with all the features found individually on other machines while remaining low in price, and never mind the impossibility of them doing so, either because of the current state of technology or because of financial considerations.

Granted, the 2500 is not a quantum leap, nor is it revolutionary, but on the other hand, the IBM, Mac, and NeXT machines aren't either. I would like to see more; more speed, more memory, more disk capacity, more colours, and so on, but I am willing to wait until Commodore can do it without going bankrupt. If some other company does it, for a reasonable price I will jump ship in a moment. Until then, I have no complaints about the state of the Amiga as compared to the offerings from other companies.

If you are one of those people I have been talking about, you'll get no sympathy from me. If you are staying with the Amiga because of some variation of "brand loyalty", I would say that you need a good lesson in logical thinking. If you are staying with the Amiga because you think it's the best machine for your purposes and within your price range, then I will say that you have no grounds for complaint. If you are not staying with the Amiga, and are going to another machine, I will wish you the best of luck, and hope that you truly know what you are doing.

AMIGA COMPUTER Center

Educational/School
Business/Medical Systems
Desktop Publishing/Desktop Video
Software Programs & Applications
Computers/Hardware/Software
Disks (Blank Media)
Fax Machines
Tutoring
Service/Repairs

Write/Call for AMIGA Product Catalog

AMIGA SPECIALISTS DISTRIBUTORS

Excellent Service and Quality

MCP Associates, Inc.

28-21 Astoria Blvd., Astoria, N.Y., 11102-1933
FAX:718-956-9028 TEL:212-221-6272
TEL: 718-956-9000

b i t s

Got an interesting programming tip, a short routine, or an unknown bit of Amiga trivia? Send it in - if we use it in the bits column, we'll credit you in the column and send you a free one-year subscription to Transactor for the Amiga.

Finding structure sizes

When testing your program for "memory leaks", you might find that it's losing say, 100 bytes each time you run it. Sounds like a structure you've allocated somewhere and never freed - but what structure is it? A list of common structures and their sizes would be handy in a case like this. There are also cases where you might want to find the size of a specific structure, like if you are declaring an array of Gadgets or IntuiTexts on the stack - just how much room are they going to take?

The following program gives a convenient way to list the size of any structure, including your own structures if you wish. If you run the program from the CLI, it will list all the name and size of all the structures it knows about; if you give it structure names as arguments, it will list only the sizes of those.

As written, the program knows about 30 of the most commonly used system structures, but you can very easily add additional structures to its repetoire. The secret to this is the INFO macro (see listing) that puts a structure's name and size into arrays. To add your own structures, just INCLUDE the header file that defines the structure, then add an INFO command for the structures of your choice. You can build a custom version of 'structsizes' for a particular project that includes all structures used by the program, and use it during debugging to track down nasty memory leaks. (If you are defining more than 200 structures, remember to change the size of the *names* and *size* arrays.)

```
/* "structsizes.c"
   Display the size of all system structures known to program, or of specific
   structures given in arguments. The INFO macro is used to make a structure
   known to the program.
*/

#include <intuition/intuitionbase.h>
#include <graphics/gfxbase.h>
#include <graphics/gels.h>
#include <libraries/dosextens.h>

#define INFO(x) { names[i] = "x"; size[i++] = sizeof(struct x); }

char *names[500]; /* known structure names */
int  size[500];   /* known structure sizes */

main (argc, argv)
int argc;
char **argv;
{
int h, a, i;

   /* define structures of interest */
   i = 0; /* point to start of structure array */
   INFO(List)
   INFO(Task)
   INFO(Gadget)
   INFO(Window)
   INFO(NewWindow)
   INFO(Screen)
   INFO(NewScreen)
   INFO(Menu)
   INFO(MenuItem)
   INFO(Requester)
   INFO(IntuiMessage)
   INFO(Message)
   INFO(MsgPort)
   INFO(IntuitionBase)
   INFO(GfxBase)
   INFO(FileLock)
   INFO(FileHandle)
   INFO(Remember)
   INFO(ViewPort)
   INFO(RastPort)
   INFO(View)
   INFO(IntuiText)
   INFO(Border)
   INFO(Image)
   INFO(Node)
   INFO(Bob)
   INFO(VSprite)
   INFO(Process)
   INFO(AnimOb)
   INFO(AnimComp)

   if (argc == 1) /* no args - list all struct sizes */
   {
      for (h = 0; h < i; h++)
         printf("Name: %-20s Size: %3d bytes\n", names[h], size[h]);
   }
   else /* list selected struct sizes */
      for (a = 1; a < argc; a++)
      {
         for (h = 0; h < i && strcmp(argv[a], names[h]); h++)
            ;
         if (h < i)
            printf("Name: %-20s Size: %3d bytes\n", names[h], size[h]);
         else
            printf("*** No info for structure \"%s\"\n", argv[a]);
      }
}
```

Find the true path, Grasshopper

The comments in the following C program - *name.c* - pretty much explain it all. Given any valid file or directory name, *name* recursively ascends the directory tree to discover the complete pathname. The program is useful in itself as a CLI tool for finding the actual location of a virtual device like "C:", but you may also want to use the routines in your own programs that need to know the actual pathname of an arbitrary user-entered file or directory. To find the parent of the current directory, for example, just use "/" as the filename argument.

```
/* name.c
   This program prints the complete path name (volume:dir/.../dir/file)
   of any filing system object. For example:

      1> name devs:keymaps/usa1
      Workbench1.2/devs/keymaps/usa1

   It may serve as the basis for a routine to be used in a file requester,
   for example, where the complete path name should be displayed to the user.
   Or it may be used in its present form for the CLI as a quick way of
   checking the true identity of a logical device.

   Return codes:
      0       Okay
      21      Couldn't get Lock on specified object
      22      Couldn't allocate memory for FileInfoBlock
      23      Examine() failed
*/
#include <libraries/dosextens.h>
#include <exec/memory.h>

BPTR Lock(), ParentDir(), Output();
void *AllocMem();
long Examine();

struct FileInfoBlock *fib;
BPTR firstlock;

main (argc, argv)
int argc; char **argv;
{
char buf[256];       /* buffer where name will be built                */
int bp;              /* pointer to next char in buffer                 */
char *filename;      /* pointer to object whose name is being determined */

   if (argc == 0) CloseStuff(20);       /* bomb if called from Workbench */
   filename = argc == 1 ? "" : argv[1]; /* use current dir if no args    */

   /* get a lock on the object, and allocate a FileInfoBlock. */
   if ((firstlock = Lock(filename, SHARED_LOCK)) == NULL) CloseStuff(21);
   if ((fib = AllocMem((long)sizeof(*fib), 0L)) == NULL)  CloseStuff(22);

   /* Build the complete path by calling ParentName. A negative return means
      an Examine() failed somewhere along the way (very unlikely)  */
   if ((bp = ParentName(firstlock, buf)) != -1)
   {
      if (buf[bp - 1] == '/')                  /* strip trailing slash  */
         bp--;
      buf[bp++] = '\n';                        /* add linefeed          */
      Write(Output(), buf, (long)bp);          /* print to stdout       */
   }
   else
      CloseStuff(23);

   CloseStuff(0);
}

CloseStuff (code)
int code;
{
   if (firstlock != NULL)  UnLock(firstlock);
   if (fib != NULL)        FreeMem(fib, (long)sizeof(*fib));
   exit(code);
}

/* ParentName
   This function receives a lock on an object, and the address of a buffer.
   Unless the lock is for a volume, it obtains a lock on the parent directory
   of the object, and calls itself with the new lock and the original buffer
   address. After it has reached the root of the filing system, it sets a
   buffer pointer to zero, and starts bubbling back up from the depths of
   recursion. At each level, it obtains the name corresponding to the lock
   argument it received, concatenates it to the buffer, adds a ':' or '/' as
   appropriate, updates the buffer pointer, and returns that to the caller.
*/
ParentName (lock, buf)
BPTR lock; char *buf;
{
BPTR parent;
int bp;
   /* if ParentDir() returns 0, we are at the root of the filing system. */
   if ((parent = ParentDir(lock)) != 0)
   {
      if ((bp = ParentName(parent, buf)) == -1) /* go one dir level up */
         return -1;
      UnLock(parent);
   }
   else
      bp = 0; /* stop recursing, start building string from buffer[0] */

   if (!Examine(lock, fib)) return -1;
   strcpy(buf + bp, fib->fib_FileName); /* concatenate name from lock */

   if ((bp += strlen(fib->fib_FileName)) == 0)
   { /* DOS 1.2 doesn't give a volume name for RAM:, so put one in */
      strcpy(buf, "RAM Disk");
      bp = 8;
   }
   buf[bp++] = parent == 0 ? ':' : '/' ;  /* ':' for root, '/' otherwise */

   return bp;              /* return updated buffer point to next level */
}
```

Make room for BASIC

Ralph L. Kohler

When there isn't enough free memory in the BASIC segment to meet your needs, (if you are working with large arrays, for instance), you can't just allocate more by putting the following statement in your program:

```
CLEAR ,70000
```

...because AmigaBasic doesn't first deallocate any memory already allocated this way. So with a 512K system this causes a BASIC "Out of Memory" error when the program is run a second time.

The cure is to see if you have already allocated extra memory before doing it again:

```
IF FRE(0) < 25000 THEN CLEAR ,70000
```

This only allocates the memory once, and there are always 70000 bytes available for BASIC thereafter.

Don't get greedy though. If your application needs things like extra screens or windows, it gets memory from the system resources, not the BASIC data segment. This means that if you have allocated too much system memory using the method described above, opening a new screen may bring about the "Out of Heap Space" message. In this case, you will have to exit BASIC, restart it, reload your program, and try again, allocating less memory this time.

Quick'n'easy filled windows

When you open an Intuition window, the interior of the window is always filled with the background colour, pencolour 0. For an attractive display, you might want, for example, a solid white window on the Workbench screen with black text and gadgets on it. You could use RectFill() after opening the window to accomplish this (taking care not to render over the title bar or sizing gadget), but there's an easy way to make the window open with any background colour you want. There's also another advantage of this technique: the background will be refilled automatically if the window is resized.

...the window will always be filled no matter how the user sizes it.

The trick is to create a dummy Gadget in the window that doesn't accept any input, but is just used to display imagery. The Gadget's associated Image structure contains no Image data, but uses the PlanePick and PlaneOnOff variables to describe a solid rectangle of any desired colour (this technique is described in the comments in the Image structure definition in the Intuition.h header file). If you make the Image the maximum size that your window will ever be, the window will always be filled no matter how the user sizes it. (This takes up no extra chip RAM for the Image). Set up Image and Gadget structures like these:

```
/* the Image describing the filled rectangle */
struct Image FillImage = {
   0, 10,        /* left, top (box starts at top left of window) */
   640, 512,     /* width, height (max window size)              */
   2,            /* Depth (2 for Workbench screen)               */
   NULL,         /* normally points to image data; none here     */
   0,            /* picks all bitplanes from PlaneOnOff below    */
   1,            /* selects the colour of the fill               */
   NULL,         /* next image structure                         */
};

/* the dummy Gadget used to render the Image */
struct Gadget FillGadg = {
   NULL,
   0, 0,             /* positioned at very top left of window    */
   0, 0,             /* size is zero, so user can't select it    */
   GADGIMAGE,        /* gadget displays an image                 */
   NULL,             /* We don't want any messages               */
   BOOLGADGET,       /* an innocuous BOOLEAN Gadget              */
   (APTR)&FillImage, /* pointer to above Image for filled box    */
   NULL, NULL, NULL, NULL, NULL, NULL
};
```

Now create a NewWindow structure as usual, putting a pointer to FillGadg in the FirstGadget structure member. If you have other Gadgets in the window, you should put the dummy gadget last on the list so that they don't get covered up by the dummy Gadget's giant filled rectangle.

Silent menus

There are times when you'd like text menu options to appear when the menu is pulled down, but not be selectable by the user. Some programmers use this technique to present information to the user, like where the program came from, but not actually receive input. In such a case, you don't want to get any messages from Intuition for the inactive menu option. If you disable the menu options, they appear "ghosted", which is not what you want.

The solution is to create a dummy menu item which is one pixel wide and disabled. The menu item should be positioned to the right of all IntuiText that makes up the displayed menu options, and it should be tall enough to accommodate all the text. In others words, the LeftEdge in the MenuItem structure should be set to a little greater than the width of the widest text displayed in the menu. The text is made with multiple IntuiText structures linked with the MenuItem; the LeftEdge for the IntuiTexts is a negative offset, relative to the MenuItem. You won't get IntuiMessages from such an item because it's disabled, and it won't be visible in the menu.

Shorter Manx C programs

If your program doesn't need to read arguments from the CLI or Workbench, you can reduce the executable size by over 700 bytes by overriding the associated compiler library functions as follows:

```
_cli_parse() {}
_wb_parse() {}
```

The linker will warn you that your new function definitions override those in the library.

Cleaning up on CTRL-C

If your Manx program does output to stdout, it can be interrupted with CTRL-C, possibly leaving resources allocated. To allow your program to clean-up before it exits (free memory, close files, etc.), create a function called _abort() (and ignore the linker warning):

```
_abort ()
{
  CleanUp();  /* your cleanup code */
  exit(1);    /* don't return      */
}
```

If you *do* return from _abort(), the program will continue as though CTRL-C hadn't been pressed.

The Changing of the Fonts

Minding your p's and q's

by Betty Clay

Betty Clay, a former mathematics teacher, chose writing for her second career. She is the Amiga columnist on STARTEXT (a videotext service of the Fort Worth Star-Telegram). She has a regular column with ICPUG, and her work has also appeared in The Amigan. Betty can be reached via CompuServe's Amiga Forum (76702,337),QuantumLink (bjc) or by mail at 1322 South Oak Street, Arlington, Texas, 76010.

Changes are coming! Versions 1.3 and 1.4 promise many changes to all parts of the Amiga. New features will bring a number of enhancements to the handling of fonts. Oh, the data will still be held in bitmaps, and there will still be CharLoc and CharSpace tables and the like. Current fonts will still work. Rather extensive enhancements are coming, however, and programmers must consider these changes and write code to make proper use of them.

What is different?

One of the most important changes is that the user will no longer be restricted to using Topaz 8 or Topaz 9 for the default font. It is true that the public domain program Setfont allowed us to install a different 8- or 9-point font instead of Topaz, but the size was predetermined. With the coming of the high-resolution monitors, this restriction would become most unpleasant, and it will not be imposed.

Until now, the programmer could depend upon the default font being of fixed width. New releases of the software will permit the user to choose a proportional font for the default. Calculating the space for a proportional font requires a different technique from that used when one may assume that a fixed-width font will be used.

Another change in 1.3 is the addition of ColorFonts. Support for these fonts was developed in cooperation with Interactive Software, publisher of the Calligrapher program. The ColorFont structure will be fully supported in version 1.4, but 1.3 will support the use of multi-colour fonts with a fixed range of colours; setting the application colour map to that intended by the designer of the font; and the remapping of one colour of the font to the RastPort APen. Features to be supported later include the remapping of the planes of the CharData with PlanePick and PlaneOnOff and the remapping of the pens of the CharData for a grey-scale font. When the ColorFont structure is fully implemented (as in 1.4), it will be supported in the Graphics library.

If you used fonts from an early version of Calligrapher, you may have found that some of them were unusable because of their memory requirements, and might be wary of using ColorFonts. (*The Amigan*, Volume III, #2 gives extensive information about this.) Newer versions of Calligrapher have this bug removed, and a public domain program called Fontrep (available on most bulletin boards) can repair whole directories of Calligrapher fonts with a single command. The problem with those fonts was caused by a misalignment in the fontname field of the program. Instead of having $0000001A in that field, the program was storing something more like $24C952 (or even larger) there. This put the name in the wrong place; the Amiga could not find the name of the font when it needed it again, so it reloaded the font, without freeing the memory from the first load. The new Diskfont library (released with 1.3) will correct these fonts as it loads them, ensuring that the colour fonts from all versions will behave properly.

These are the three major changes, then. The availability of many disks full of fonts, the introduction of high-resolution monitors, and the numerous programs that make use of colour fonts have made the changes necessary and desirable. How will the changes affect you?

Caution!!

There are several things the programmer must do in order to make certain that programs will continue to work well and to look good under the new circumstances. Here are the major considerations:

1. Programs must gracefully handle large numbers of fonts

Many programs currently in use are breaking because of this flaw. Probably the most notorious is the popular DPaint. It always gurus when given a long list of fonts, and this is a very serious shortcoming. The use of a variety of fonts is important

in many DPaint applications, but the program is very limited in the number of fonts it can access at one time, and offers no facility for reading a different Fonts: directory.

An even worse case is found in DeluxePrint. It allows only the use of the fonts available on the original Workbench. How sad it is that two programs that are so useful would have been so short-sighted on this particular point. They were, however, among the earliest Amiga software offerings. For a programmer to duplicate these shortcomings now is inexcusable.

2. Programs must be able to handle various sizes of fonts

When the programmer tests his program using the current default fonts, it may look great. When his customer uses that program on a high resolution monitor such as the A2024, it might be unreadable. The new software will allow the user to select his own default font. Without care on the part of the programmer, the font the user selects may not fit into the space allotted for the necessary text. With care, the software can adjust to this flexibility without sacrificing any of the desired effects.

3. From the very beginning, Commodore-Amiga has stressed that programmers should follow the rules set down in the RKM

When those conventions are followed, the new software releases will provide all the protection necessary to ensure that programs continue to work. Some programmers like to find neat ways to get around the rules, and their programs will experience difficulty when the system software changes. This applies to fonts as well as to all other parts of the Amiga.

From the very beginning, Commodore-Amiga has stressed that programmers should follow the rules set down in the RKM...

Finding the information

Most text is rendered through Intuition, so it is good to know where Intuition looks for the fonts it uses. Usually, the text to be rendered is part of an IntuiText structure, and it uses the font specified in the ITextFont TextAttr. IntuiText will use a font from one of three places:

Screen->Font,
Screen->RastPort.Font, or
Screen->BarLayer->rp->Font.

When a screen is first opened, all three of these will refer to the same font and that will be the one specified in the NewScreen structure if it was available and openable; otherwise, the default font will be used. It is possible to change any of these three fonts as the program runs, but if one of them is changed, all three should be changed.

Intuition renders most of the text we see on screen, such as: Gadget text, highlighted text and the text in string gadgets; menu titles, items and subitems; window and screen titles; and the text in requesters and alerts. Changing one of the fonts used by Intuition can cause problems in any of these areas, and this is especially likely if they are not all changed to the same font when a change is made. A particular instance is that the screen and window title bar heights are determined by the height of the font to be used in them. Making careless changes to the height of that font as the program runs will cause noticeable problems.

Of all the things one should remember about Intuition and fonts, probably the most important is that Intuition will never call for a font from disk. If the desired font is not already loaded into memory and available for use, Intuition will go to the default font. If a diskfont is desired then, it is necessary that the programmer open the diskfont before calling Intuition, leave it open until it is no longer needed, and then close it before leaving.

Calculations for text

Since users will soon be able to select fonts to suit their particular preferences and requirements, and since the programmer cannot possibly know what characteristics those fonts will have, there must be ways to make the software adjust to any font that might be selected.

One way, of course, is to specify a particular font to be used and to make certain that the specified font will be available to the program when it is needed. The user may not like the font that was chosen. On his screen, it may be completely unreadable. Providing and specifying a particular font does allow the programmer to make certain that there is enough space for the menus, gadgets, etc., to render their text.

It is a far friendlier thing to let the user choose the font he wants to use, and then ask the computer to tell you the characteristics of that font. The height of his font will be stored in the TextFont and RastPort structures, and the programmer can just access the proper field to get the font height. To get the width of a string, the IntuiTextLength() function is said to be easier to use but, since the TextLength() function will report the length of the string in pixels, this is a more accurate method of determining string width.

It is even possible to set up a special RastPort just for making text calculations. Before opening a custom screen, the programmer can use InitRastPort() and SetFont(), and can then safely calculate the title bar heights, widths, and layouts for all of the things on his screen. This method will work for a user-specified font, a programmer-specified font, or for the default font that is being used. (Information about the default font may be obtained by referring to GfxBase->DefaultFont.) From Workbench screens, the information can be obtained from GetScreenData.

Other changes will follow these. New structures will be added to the system software that will make it easier to calculate the space needed for text using unknown fonts. It is quite likely that we soon will have separate files for bold and italic fonts, and that there will be good ways to scale existing fonts automatically with less distortion than we now find. Also, the Amiga is being sold in many nations. To ensure that a program is usable in foreign countries, the fonts we use must be able to adapt to the languages of those nations, and must contain the characters needed to write in those languages.

Changes are already here, and even more changes will follow. The Amiga just keeps getting better.

The font content files

See *Transactor for the Amiga*, Volume 1, Issue 1 if you are not familiar with the font content header files - the ".font" files. They contain a file ID ($0x0F00), the number of entries for this particular family of fonts, 256 bytes to hold the path needed to locate the actual font, and the height, style, and flags for the font. If there is more than one size or style in the family, all except the file ID will be repeated for each entry. These are very short files, and were designed to speed up the locating of the desired font.

It was not until recently that anyone really explored these files and made proper use of them. Stephen Vermeulen (author of Express Paint), recently posted to Usenet and other bulletin boards a program he calls REN. It provides one of the most helpful font utilities yet to appear, and Bob Burns of Commodore was quick to assure Stephen that it will not only continue to work under version 1.3, but that the utilities in that release will not duplicate his achievement. Steve's work deserves a very wide audience.

Using REN and some of the fonts I have accumulated, I recently prepared two disks of fonts containing only the actual font data, with the FontContentsHeader files remaining in the Fonts: directory of my boot disk. When all of the data files were removed to another disk, there was room for more than fifty font names in this directory, with many of those having many sizes. The data for those fonts was stored on two disks, and each disk was divided into several directories. Pagesetter and Notepad had no difficulty locating any one of those fonts whenever I called for it. DPaint gurued.

So how did Steve do it? He simply used those 256 bytes for their avowed purpose. You tell his program where you intend to store the data for a font, and his program writes that path into the FontContentsHeader file. It does this for each size and style at the same time. Then, when you call for a font, this path is read and, if the Fonts disk needed is unavailable, the Amiga will put up a standard requester saying "Please place Volume xxxxxx in any drive". Since the proper disk is named by the requester, I can confidently slip in the correct disk and everything works as smoothly as if the fonts were all still crowding my boot disk. What a wonderful tool! Thanks a million, Steve!

Here, with Stephen's permission, is the program:

```
/*****************************************************
  ren.c

  This program allows one to rename the directory
  which contains the font data files.

  Copyright 1988 By Stephen Vermeulen

  This program may be freely redistributed so long as
  the Copyright messages remain intact and the program
  is distributed with the supplied documentation.

  syntax:

     ren thisfont.font path
*****************************************************/

#include <stdio.h>

char newname[256], oldname[256];

main(argc, argv)
int argc;
char *argv[];
{
  FILE *font;
  long j, pos;
  short i, n;

  if (argc != 3)
  {
    printf("ren filename.font newpath\n");
    printf("Copyright 1988 By Stephen Vermeulen\n");
    printf("3635 Utah Dr. N.W.,  Calgary, Alberta, CANADA, T2N 4A6\n");
  }
  else
  {
    font = fopen(argv[1], "r+");
    if (font)
    {
      /*******************************************
        now determine how many sizes this font has
      *******************************************/

      fread(&n, 2, 1, font);  /** throw away **/
      fread(&n, 2, 1, font);  /** number of sizes **/
      printf("Font %s has %d sizes\n", argv[1], n);
      for (i = 0; i < n; ++i)
      {
        /** for each size we rename it...
         **/

        fseek(font, 4L + 260L * i, 0);
        fread(oldname, 256, 1, font);
        printf("name was: %s ", oldname);
        for (j = 255; j > 0; --j)
          if (oldname[j] == '/') break;
        strcpy(newname, argv[2]);
        strcat(newname, oldname + j);
        fseek(font, 4L + 260L * i, 0);
        printf("now is: %s\n", newname);
        fwrite(newname, 256, 1, font);
      }
      fclose(font);
    }
  }
}
```

Calling the CLI from AmigaBASIC

"...the whole machine at your fingertips."

by Jim Butterfield
Copyright © 1988 Jim Butterfield

Jim Butterfield is a Toronto writer, programmer and lecturer whose association with microcomputers goes back to the 1K KIM-1 in the earliest days of Commodore computing. Jim's ability to communicate his encyclopedic knowledge of CBM products through articles, books, lectures and television programs has made his name a household word among Commodore users around the world.

AmigaBASIC is a fast and powerful language, supplied with every Amiga. It has a large vocabulary and does many jobs. But sometimes you might like to do something that's easier or more natural in the CLI - the Command Line Interpreter. This article will show you how.

Almost any program that can be run on the Amiga can be called in by the CLI. So if you can get AmigaBASIC to access the CLI, you'll have the whole machine at your fingertips. But we'll need to learn a few rules.

Tapping the library

To connect to the power of the Amiga operating system, you need to communicate with the system *libraries*. Some of these are in the Kickstart ROM, some are in RAM, and some are on disk. You don't need to worry about where they are. All you need is to get three elements together.

The first item is BASIC's LIBRARY statement. When you command LIBRARY followed by a library name, BASIC makes the connection to the inner workings of the Amiga. In this series of activities, we'll make use of the DOS (Disk Operating System) library. Our command will be: LIBRARY "dos.library".

As we move towards the Amiga's operating system, we'll need to develop a new habit: case consciousness. In 'regular' AmigaBASIC, we can mix upper and lower case at will. It won't matter if we command "print" or "PRINT" or, for that matter, "PrInT". BASIC knows what we mean. Similarly, we can call a variable "subtotal" or "SubTotal"; there's no confusion and it will be taken as the same variable in either case. File names don't seem to care about case, either. But the operating system is case sensitive; follow the upper and lower case usage in the listings carefully.

We connect to a library with the LIBRARY function, and when we're finished with it, we disconnect with LIBRARY CLOSE. More than one library can be active at a time. But to allow LIBRARY to work correctly, we need another item: the *.bmap* file. We'll talk about this in a moment. If you don't have the appropriate *.bmap* file, you'll have to make one.

If we want to call *subprograms* in the library, the LIBRARY command (with the appropriate *.bmap* file in place) is all that's needed. But if we want to call *functions*, we have another command that we must call: DECLARE FUNCTION (followed by the function name and then the word LIBRARY).

Subprograms vs. functions

A *subprogram* is much like a subroutine. You call it, and it does something. You might compare it to a call such as "GOSUB DingDong"... you go somewhere, do a job, and then come back. More accurately, we might command "CALL xClose"... but the idea is the same.

A *function*, on the other hand, calculates a value. If you wanted to calculate a square root, you'd use the normal BASIC function SQR(). You never use a function by itself... it's part of an expression, such as PRINT SQR(16)+SQR(25).

A library function must be declared before you use it, with DECLARE FUNCTION. The most visible purpose of such a declaration is to identify the *type* of value that will be returned by the function. The type is almost always a *long integer*, which is signalled by using the character '&' after the function name. Thus, before using a function called xOpen, we would code:

```
DECLARE FUNCTION xOpen&() LIBRARY
```

You might think, based on past experience with BASIC, that subprograms are for doing things and functions are for calculating things. Not so. In fact, functions are used for almost everything in the Amiga. Here's how it works.

Let's say you want to do something like open an output channel. Sounds like a subprogram, right? Nope. It's a function. We might code:

```
h& = xOpen&(stuff about the channel)
```

Now, when we call the function **xOpen&** (watch out for the upper and lower case), it opens a channel as we requested, and then returns a channel identity to use (called a *handle*) which we may use any time we need to reference that channel. If for some reason it can't open the channel, it returns a value of zero. So we could follow the above code with:

```
IF h&=0 THEN PRINT "Can't Open the channel..":STOP
```

If the handle is non-zero, we have opened the channel successfully. The handle, by the way, contains an address; we'll deliver that address to any other activity that needs to use the channel. But you don't need to worry about that, either. Just follow the rules and everything will work.

The *.bmap* file

Go into CLI, and command: "DIR LIBS:"... you'll see several items stored in this directory. You *might* see a file called *dos.bmap*, but probably not. If you have such a file, skip to the next section. If you don't, we'll have to make one.

Appendix F of your AmigaBASIC manual contains a lot of obscure material about *.bmap* files. We'll talk about this in detail later. For now, only the last sentence is important, and here's what it tells you: the best way to get a *.bmap* file is to dig out your EXTRAS disk (the one with AmigaBASIC), find your way into the BasicDemos section, and run the program *ConvertFd*.

If you do so, remember that the name of the file you want to convert is called DOS. Plough your way through it, and after a few false starts, you will likely end up with a file named *dos.bmap*. Or... I hate to spoil your fun, but *dos.bmap* is probably already on that disk, in the BasicDemos library.

In any case, the next thing to do is to copy it over so that it's also in LIBS:, where it can happily live for ever. You can do that with the CLI command:

```
COPY EXTRAS:BasicDemos/dos.bmap LIBS:
```

If all that sounds too complex, you could type in Program 1, Bmap_Maker, which will make a 'tiny' version of *dos.bmap* for you. *[Note: BMap_Maker and the other programs accompanying this article are also available on the disk for this issue. - Ed.]* Run the program and it will create the file. There will also be a file called *dos.bmap.info* which is not needed; you may delete this if you wish.

Getting to CLI

Now that you have *dos.bmap* safely installed in your LIBS: library, we're ready to see the trick done. Here's the secret.

Command **Execute** within the DOS library will execute any CLI command you give it. That's where BASIC truly gains access to the whole machine. What we will do is to cause **Execute** to give the SORT command, allowing us to sort a file quickly.

Program 2, *SortCLI*, allows you to input names. The names are delivered to RAMdisk as they are received... the file is called TEMP0. When you're finished, the file is closed and the interesting work begins.

We open the LIBRARY. We define the function **Execute&**. Note that it is a function, not a subprogram; it returns a value (a boolean 'success' flag).

Now we **Execute** the command SORT TEMP0 TEMP1 and wind up our library activities. All that remains to do is to pull the sorted values out of TEMP1 and print them for your enjoyment.

Cautionary Note

This version of the program runs only if you call AmigaBASIC from the CLI. That is, you start up a CLI and command "AmigaBASIC" or "Run AmigaBASIC".

If you start up AmigaBASIC from the Workbench, clicking over the program icon or the AmigaBASIC icon, the program will crash dismally.

To find out why, and how to fix it, stay tuned...

Program Details

It might be worth looking through the code of the SortCLI program to see how the call to CLI works. Much of the program is "regular BASIC." For example, there's code at the beginning to allow you to enter data, which is immediately passed to a RAM file. There's code at the end to print the contents of the sorted RAM file. Both of these are straight BASIC.

The code that links to CLI may be found in the middle. The first instruction to do the job reads:

```
DECLARE FUNCTION Execute& LIBRARY
```

This line says, "When I use a *function* called **Execute**, I want the computer to know that it's a library function, and that the value returned by the function will be a *long integer*".

As we noted earlier, **Execute** is not a *command*, but a *function* - it returns a value. The value is *boolean*, that is, it signals either *true* or *false*. In this case, it signals "operation successful" if true.

The place to read up on all of this is in *The AmigaDOS Manual*. Within the section entitled *AmigaDOS Developer's Manual* you'll find "Calling AmigaDOS" (Chapter 2), and that chapter has an entry for the **Execute** function in Section 2.2 under the heading *Loading Code*. The form is shown as:

```
Success = Execute( commandString, input, output )
```

We'll set the command string, set input and output to zero, call the **Execute** command, and will receive back a value for *success*. Soon. But first, we have to open the library itself with:

```
LIBRARY "dos.library"
```

Now for the actual call to the CLI:

```
x=Execute&(SADD("sort >nil: ram:temp0 ram:temp1"
  +CHR$(0)), 0, 0)
```

The command string is enclosed in quotes, and must be followed by a NULL character; that's how the system detects the end of the command. We supply its address. We have set *input* and *output* to zero. The call to **Execute** will return a "success" signal, which we'll put into variable *x*... and never check.

Now that we've done the job, we wrap up with:

```
LIBRARY CLOSE
```

From this point, we continue with 'regular' BASIC.

Where's the bug?

That's all very well, but why won't this program work from the Workbench? Let's read further in the documentation for **Execute**:

"...in most cases the output file handle must be provided, and will be used by the CLI commands as their output stream unless redirection was specified."

Well, SORT doesn't need an output stream. We tell it where to put the sorted file. Let's keep reading:

"...If the output file handle is set to zero then the current window, normally specified as *, is used. *Note that programs running under the Workbench do not normally have a current window.*"

There it is. We need an output window, and the Workbench doesn't provide it. Even though we never send anything to the output window, it must be there.

So let's set up an output path... even though it will never be used. I'll set up an output to NIL:... whatever goes there will be thrown away. This is a type of double negative: we're sending nothing, and throwing it away when it gets there. But it satisfies the need for an output path.

Opening the path

In the same documentation, you'll find details on the **Open** function (which, you'll remember, returns a value called a *handle*, which we'll use). We'll use it to open a path to NIL:... later, we'll wind up the path with subprogram **Close**. Note that **Close** is *not* a function, and we don't need to declare it.

The format for **Open** is:

```
handle = Open( name, AccessMode )
```

The *name* is a string, and we'll handle it the same way as we did for **Execute**, that is, we'll terminate it with a zero and supply the address of the string, not the string itself. The *AccessMode* has only three useful values: 1004 for opening a file for read and write; 1005 for opening a file for read; and 1006 for opening a new file for write. And *handle* is the value we'll get back, allowing us to identify the file. It will be a *long integer*, so we must remember that when we define the function.

So here's the plan: We must remember to DEFINE FUNCTION for **Open** (it's in the same library as **Execute**). When our call to **Open** gives us the file handle, we'll supply that handle to **Execute**. Now that function has an output path, it will be happy even when called from Workbench.

One small problem: the file *dos.bmap* does *not* contain a function called **Open**. Instead, it has a function called **xOpen**. That's for a good reason: OPEN is a BASIC keyword, and AmigaBASIC would be confused if we used it. So we'll use **xOpen** (and, later, **xClose**), but we'll drop the 'x' when we read the documentation.

So we're ready to type in Program 3, *SortWB*. This time, it will do the job no matter how we called AmigaBASIC.

The .bmap files

Now it's time to look more carefully at the *.bmap* files, which hint at the workings of the Amiga, and to put together a program showing the variety of ways we can use CLI to do jobs for AmigaBASIC.

In some ways, the structure of *.bmap* files reflects the structure of the Amiga itself. It's worth a closer look, and the program *Bmap_Snoop* (Program 4) will do the job for you.

Library structures

The Amiga contains a number of structures called *libraries* that make the inner workings accessible to programmers. These libraries contain a number of interesting things, but the part we're interested in right now is called a *jump table*. A jump table is a series of 'jump' instructions that tell the computer to go to a certain place to do a certain job.

Each library has a different jump table, customized for its work. The DOS library, for example, contains the entries we have already - **Open**, **Close** and **Execute** - plus many others such as **DeleteFile**, **Read**, and **Lock**. Each of these functions occupies its own special slot in the jump table.

A library has a 'reference point', or address, which locates it, but this address is not at the start of the library structure. Addresses *above* the reference point usually hold data of some

sort. Addresses *below* the reference point contain the jump table. So don't be surprised if a jump table entry is shown at location -60 in its library... that's where such entries always lie.

So what do we need to know, in order to make the connection? We need the name of the library (in our example, "*dos.library*"). Each library has its name built in. So a command such as "LIBRARY "*dos.library*"" will cause BASIC to ask the operating system to look for the library of that name and report its location. Yes, it can move around - but not while you're using it.

If we've found the library, thanks to the LIBRARY command, we must now find the function we want to use: **Open**, **Execute**, or whatever. That's an offset within the library... but which is which? File *.bmap* comes to the rescue!

The *.bmap* file contains the names of entries into a given library, and the 'offset' for that particular function. The offset is always negative. There's one other piece of information in the file, but we'll come to that in a moment.

Function **Open**, for example, is first in the *dos.bmap* file, and is shown to have an offset of -30. Function **Execute** is last in the file, and has an offset of -222 (it's a big jump table).

With the name of the library and the *.bmap* file, we can go directly to the part of the jump table that we want. There's only one more thing we must sort out: how much data must we pass to a function? Some functions require no data... others need a whole collection of information, and this information must be carefully arranged in the 'registers' of the processor at the time the function is called. How can a BASIC program sort out what's needed?

Again, *.bmap* files provide the answer. In addition to the offset value for the function, the file also carries information on how the data should be set up in order to make the function behave properly. A list of registers is supplied to indicate to the interface what information should be provided where. The registers may be marked 'D' for Data register, or 'A' for Address register, followed by a number from 0 to 7.

Don't worry about what the register does. The computer will take care of that. You'll find it useful to know that a register may contain an integer value of up to many millions, or a memory address. If a number can do the job, you'll likely use it directly; if a string is called for, you'll supply its address (and terminate the string with a NULL character to signal its end).

Program Bmap_Snoop

Enter and run this program (Program 4). It prompts for the name of some popular *.bmap* files. You may select one of these, or enter your own name.

The program will print out the names of the functions within the library, the offsets, and the registers that are needed. That's all there is... no secret code, just a simple table.

But you will notice that there are *lots* of functions. It takes a lot of time, and a lot of reading of manuals, to gain an appreciation of what they all do.

Program DirMake

To tie this together, I've written program *DirMake* (Program 5) to show several ways of using the BASIC interface to CLI. The program does something that's not easy to do in BASIC... it creates a new directory.

The first call to **Execute** invokes the CLI command INFO. The output is delivered to file *RAM:temp*, and it's used to spot the currently active devices (disks, RAM, whatever). Select the one you want.

For the first call, we include the CLI command right in the **Execute** line.

The second call, which may be repeated, looks at a given directory, using the CLI command DIR, to see if there are any subdirectories within it. If there are, it allows you to choose one if you wish. If you do so, this section will repeat.

For the second call, we form the command string separately and then reference it as a variable from the **Execute** command.

The third call uses the CLI command MAKEDIR to make the desired directory. For the sake of variety (and to show you that it can be done) we put this command into a script file, and ask **Execute** to run the commands from the file. There's only one in this case, but a whole series of commands could be executed if desired.

Summary

Using the magic of commands LIBRARY and DECLARE FUNCTION, and the elegant *.bmap* file, we can add a great deal of variety to BASIC.

And in doing so, we can also gain more insight into the workings of the Amiga.

Program 1: Bmap_Maker

```
REM   Bmap-maker       Jim Butterfield

DATA 120,79,112,101,110,0,255,226,2,3
DATA 0,120,67,108,111,115,101,0,255,220
DATA 2,0,69,120,101,99,117,116,101,0
DATA 255,34,2,3,4,0

FOR j=1 TO 36
 READ v
 t=t+v
 m$=m$+CHR$(v)
NEXT j
IF t<>3128 THEN PRINT "DATA ERROR":END

PRINT "making file <Libs:dos.bmap> ..."
```

```
 OPEN "libs:dos.bmap" FOR OUTPUT AS #1
  PRINT#1,m$;
 CLOSE #1
PRINT "file <Libs:dos.bmap> written"
```

Program 2: SortCLI

```
REM - This demo program shows how to
REM - invoke Amiga EXECUTE routine
REM - from Amiga-BASIC

PRINT "Name sorter demo ... Jim Butterfield"

PRINT "Enter a number of names.."
PRINT "Finish with a blank line."
OPEN "ram:temp0" FOR OUTPUT AS #2
n=1
PRINT "Name";n;":";
LINE INPUT n$
WHILE n$<>""
 PRINT#2,n$
 n=n+1
 PRINT "Name";n;":";
 LINE INPUT n$
WEND
CLOSE #2
PRINT
PRINT "sorting names...";
DECLARE FUNCTION Execute& LIBRARY
  REM --- The next line only works
  REM --- under CLI, not WorkBench
  LIBRARY "dos.library"
  x=Execute&(SADD("sort >nil: ram:temp0 ram:temp1"+CHR$(0)), 0, 0)
  LIBRARY CLOSE
PRINT "... sort complete."
PRINT "   Here are the names:"
OPEN "RAM:temp1" FOR INPUT AS 1
WHILE NOT EOF(1)
  LINE INPUT #1,a$
  PRINT a$
WEND
CLOSE
KILL "RAM:temp0"
KILL "RAM:temp1"
PRINT
END
```

Program 3: SortWB

```
REM - This demo program shows how to
REM - invoke Amiga EXECUTE routine
REM - from Amiga-BASIC

PRINT "Name sorter demo ... Jim Butterfield"

PRINT "Enter a number of names.."
PRINT "Finish with a blank line."
OPEN "ram:temp0" FOR OUTPUT AS #2
n=1
PRINT "Name";n;":";
LINE INPUT n$
WHILE n$<>""
 PRINT#2,n$
 n=n+1
 PRINT "Name";n;":";
 LINE INPUT n$
WEND
CLOSE #2
PRINT
PRINT "sorting names...";
DECLARE FUNCTION xOpen& LIBRARY
DECLARE FUNCTION xClose& LIBRARY
DECLARE FUNCTION Execute& LIBRARY
 '  This works with WorkBench or CLI
 '  cause it creates its own NIL: output
 LIBRARY "dos.library"
 z$=CHR$(0)
 handle&=xOpen&(SADD("NIL:"+z$),SADD("MODE_NEWFILE"+z$))
 IF handle&<>0 THEN
  x=Execute&(SADD("sort ram:temp0 ram:temp1"+z$), 0, handle&)
  xClose(handle&)
 ELSE
  PRINT "couldn't handle file i/o"
 END IF
 LIBRARY CLOSE
PRINT "... sort complete."
PRINT "   Here are the names:"
OPEN "RAM:temp1" FOR INPUT AS 1
WHILE NOT EOF(1)
  LINE INPUT #1,a$
  PRINT a$
WEND
CLOSE
KILL "RAM:temp0"
KILL "RAM:temp1"
PRINT
END
```

Program 4: Bmap Snoop

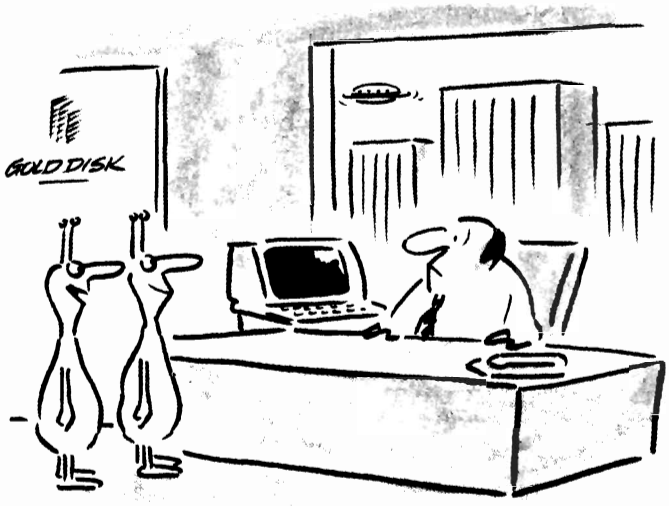
```
PRINT " BMAP Looker   -   Jim Butterfield"
PRINT
DATA diskfont,dos,graphics,exec,other,end
READ f$
f=0
WHILE f$<>"end"
 f=f+1
 IF f$="other" THEN
  PRINT f;" other"
 ELSE
  PRINT f;" :libs/";f$;".bmap"
 END IF
 READ f$
WEND
ch=0
WHILE ch<1 OR ch>f
 INPUT "Your choice";ch
WEND
IF ch<f THEN
 RESTORE
 FOR j=1 TO ch
```

```
  READ f$
 NEXT j
 nam$=":libs/"+f$+".bmap"
ELSE
 PRINT "Enter full .bmap file name: ";
 LINE INPUT nam$
END IF
OPEN nam$ FOR INPUT AS #1
 bfile$=INPUT$(LOF(1),1)
CLOSE #1
sName=1
n$=CHR$(0)
WHILE sName<LEN(bfile$)
 eName=INSTR(MID$(bfile$,sName),n$)
 PRINT MID$(bfile$,sName,eName-1);TAB(25);
 ofset=CVI(MID$(bfile$,eName+sName,2))
 PRINT USING "#### ";ofset;
 sep$=" "
 parmPt=eName+sName+2
 parm=ASC(MID$(bfile$,parmPt))
 WHILE parm<>0
  parm=parm-1
  IF parm>7 THEN
   handle$="A"+CHR$(48+parm-8)
  ELSE
   handle$="D"+CHR$(48+parm)
  END IF
  PRINT sep$;handle$;
  sep$=","
  parmPt=parmPt+1
  parm=ASC(MID$(bfile$,parmPt))
 WEND
PRINT
sName=parmPt+1
WEND
```

Program 5: DirMake

```
REM - This demo program shows how to
REM - invoke Amiga EXECUTE routine
REM - from Amiga-BASIC
DIM x$(20)
PRINT "Let's make a directory..."
DECLARE FUNCTION xOpen& LIBRARY
DECLARE FUNCTION Execute& LIBRARY
LIBRARY "dos.library"
z$=CHR$(0)
Hand1&=xOpen&(SADD("NIL:"+z$),1006)
x=Execute&(SADD("info >ram:temp"+z$), 0, Hand1& )
 OPEN "ram:temp" FOR INPUT AS #1
  PRINT 0;"< none >
  WHILE NOT EOF(1)
   LINE INPUT #1,x$
   IF MID$(x$,4,1)=":" THEN
    n=n+1
    x$(n)=LEFT$(x$,4)
    PRINT n;x$(n)
   END IF
  WEND
 CLOSE #1
KILL "ram:temp"
p=-1
WHILE p<0 OR p>n
 PRINT "your choice (0 to";n;")";
 INPUT p
WEND
 IF p<>0 THEN
  WHILE p<>0
   File$=x$(p)
   PRINT
   Com$="DIR >ram:temp "+File$+" opt d"+z$
   x=Execute&(SADD(Com$), 0, Hand1& )
   x$(0)=File$
   OPEN "ram:temp" FOR INPUT AS #1
   n=0
   WHILE NOT EOF(1)
     LINE INPUT #1,x$
     n=n+1
     x$(n)=File$+MID$(x$,6,LEN(x$)-11)+"/"
   WEND
   CLOSE #1
   KILL "ram:temp"
   IF n>0 THEN
    PRINT 0;File$;"   ... or,"
    FOR j=1 TO n
     PRINT j;x$(j)
    NEXT j
    p=-1
    WHILE p<0 OR p>n
     PRINT "your choice (0 to";n;")";
     INPUT p
    WEND
   ELSE
    p=0
   END IF
   File$=x$(p)
  WEND
  PRINT
  PRINT "We're going to put a new Directory (Drawer)"
  PRINT "into area   ";File$
  INPUT "OK";x$
  IF x$="y" OR x$="yes" THEN
   INPUT "name of new directory";Nam$
   OPEN "ram:temp" FOR OUTPUT AS #1
     PRINT #1,"MakeDir ";File$;Nam$
   CLOSE #1
   PRINT "Making directory ";File$;Nam$
   ' Now we use a script file...
   Hand2&=xOpen&(SADD("ram:temp"+z$),1005)
   IF Hand2&<>0 THEN
    x=Execute&(SADD(z$), Hand2&, Hand1& )
    xClose(Hand2&)
   ELSE
    PRINT "Failed!  Couldn't handle file i/o"
   END IF
  END IF
  KILL "ram:temp"
  FILES File$
 END IF
xClose(Hand1&)
LIBRARY CLOSE
```

We're looking for some far out thinkers!



"We're far out and very bright!"

You don't have to be from outer space. You just have to be a unique computer programmer.

If you have an Amiga program near completion that you think is a winner, you should contact Gold Disk right away. We're looking for top quality software to market. Working with us could be the most rewarding move of your life.

Write or call us for our free developer's kit, "Hitch your wagon to a star."

GOLD DISK
P.O. Box 789, Streetsville, Mississauga
Ontario, Canada L5M 2C2
Telephone: (416) 828-0913

DIGITAL DEPOT

world of commodore
BOOTH #123

DIGITAL DEPOT is a service bureau that allows a client with just an Amiga 500 the ability to produce professional results in graphics, sound, video, film and even the production of solid three dimensional prototypes to tolerances of .005 inches. If you can produce something on a computer, you probably will be able to use the output services of DIGITAL DEPOT.

Not all DIGITAL DEPOT locations will have the ability to produce all of the services. While many will be full service, many will start with only one or two services on-line and will forward all other work to full service centers. A DIGITAL DEPOT may only do video on-site, for example.

If you want more information on how you or your dealership can become a DIGITAL DEPOT, whether you want to just have a color scanner on location or the works, please leave your card with one of our representatives at our booth, and we will be in contact with you. To stay in touch, logon to the DIGITAL DEPOT bulletin board at 1-201-864-0376.

DIGITAL DEPOT

√√√√ UEDIT √√√√

Programmable Editor for Technical (♂♀ ♥♪♫ ☼☺) Users

Uedit has quietly become the Editor Of Choice for bright programmers and professionals in the USA and more than a dozen countries. Recently it won the Gold Star in Commodore magazine.

FEATURES: Ultra power, capacity, and flexibility. **Uedit** has a Rexx interface, many wordprocessing features, optional spelling checker, and a truckload of editing features. How do you describe a program that lets you add new features when you feel like it? Yet you can use it right away "as is". Edit 20, even 100 or more files. Auto-load/edit/save a batch of files while you relax. Every command and feature is customizable, even the online Help and Teach Keys facility!

USER SURVEY: 37% have a doctorate in physics, CS, math, engineering, medicine, etc. 82% use **Uedit** for programming. 73% use it for wordprocessing. 75% use the standard config "as is" or modify it slightly. 25% configure **Uedit** to emulate a popular program or to create their preferred environment.

FEEDBACK: "the best editor anywhere, anytime, period." "the best editor on ANY computer." A BYTE technical editor began his registration letter with: "Great duck! What an editor!" About the new 173 page printed manual: "a showpiece." "well written and organized." "I love it!"

Uedit is $59 + $3 shipping. (Foreigners add $3). It includes:
- * Fully customizable **Uedit** plus shareware version
- * Professional quality 173 page printed manual
- * $15 shareware commissions for finding new users
- * Mailings, notice of upgrades, user assistance

Options:
- * Spell-checking **Uedit** & dictionary utilities, $15
- * Quarterly Newsletter, $12 per year

Rick Stiles, P O Box 666, Washington, IN 47501 (812) 254-4986

InterComputing Inc.

2100 N. Hwy 360, Suite 2101
Grand Prairie, TX 75050-1015

Orders only call : 1-800-531-4747

Customer Service
and TX-Orders call 1-214-988-3500

Amiga is a trademark of Commodore-Amiga, Inc.

AMIGA

We are in business since 1984;
We support the AMIGA since 1985;
and we will give you the best service possible at prices the other 'Discount-places' wont like - PERIOD !
Please call for a complete listing our products !

HARDWARE

Memory

8-Up Memorycard OK	145.75
8-Up Memorycard with 2MB	649.00
8-Up Pop-Simm 256k or 1MB	17.95
Starboard 0 MB A1000	239.75
Starboard 0.5MB A1000	399.75
Starboard 1 MB A1000	599.75
Starboard 0 MB A-500	239.75
Starboard 0.5MB A-500	399.75
Starboard 1 MB A-500	599.75
Starboard Upper Deck ok	44.75

Drives

FData-10 ext. Drive	139.75
int. 3.5" Drive A2000 only	145.75
HardFrame SCSI Card	239.75
Starboard SCSI Module	97.75
Supra 20MB HDisk	649.00
Supra 30MB HDisk	749.00

HARDWARE of the Month

Flexible Data Systems

FDATA-10 Disk Drive

This external 3.5" Drive is a very inexpensive unit to make the life with your Amiga easier.
The FData-10 is compatible with the original Amiga-drive but is quieter, smaller and has a longer cable (27inch) to connect it to the Amiga.

Oh yes, before we forget to tell you,
it's only $139.75 !

Video / Graphics

A-Pro Draw 12x12	499.75
A-Pro Draw 9x6	399.75
Amiga Live A 500	299.00
Amiga Live A 2000	329.00
Amiga Live A 1000	219.00
Digi View 3.0 NTSC	139.75
Digi View Adapter	19.75
Digi Droid	64.50
Easyl 500	299.75
Easyl 1000	349.00
Easyl 2000	349.00
Flicker Fixer	499.00
Flicker Master	12.50
Frame Grabber	549.75
Gen One Genlock	749.75
Imprint	399.75
Light Pen	99.75
Mitsubishi AUM 1371A	549.00
Panasonic 1410 Video Kit	259.75
Perfect Vision	175.00
Perfect Vision Color Splitter	85.75
Polaroid Palette	call
Pro Gen	375.00
Super Gen	699.75
X-Specs 3D	97.75

Great Hardware Buys

We now carry a wide variety of
MIDI-Equipment

Synthesizer	
Kawai K1 Synthesizer	749.50
Yamaha FB-01 Module	329.50
Percussions	
Alesis HR-16 Drum machine	379.50
Effects	
Alesis MIDIVerb II Digital Reverb	249.50
Mixer	
Kawai MX-8R Mixer (8 Channel / 2Effects I/O)	279.50
Midi	
ECE-Midi Interface	44.75
Gold Midi internal for A2000	69.95
Midi Cable	8.75

Sound / Midi

ECE-Midi w/ RS-232 thru	44.75
Future Sound Sampler	129.75
Soundscape Sampler	79.50
Soundscape Midi Interface	42.75
Perfect Sound A500/2000	59.75
Perfect Sound A1000	59.75
Midi Cable	8.75
Midi Gold for A2000 internal	69.95

Miscellaneous

Fuji 3.5" DS/DD Disks	17.75
Mouse Pad	6.95
Mouse Time for A1000	29.75
Multifunction Card	69.75
Phasar Gun	46.75
Processor Accelerator	149.50
Sony 3.5" DS/DD Disks	19.50
Supra Modem 2400 Baud	149.75

DESKTOP PUBLISHING

Comic Setter	66.75
Font Set I	24.75
Laserscript	29.75
Newsletter Fonts	29.75
PCM-Fonts Vol.1-3	27.50
Page Setter	89.75
Professional Page	239.75
Shakespeare	149.75
Studio Fonts	29.75
Zuma Fonts 1,2,3,4 each	24.75

MUSIC

Audio Master	39.75
Deluxe Music	59.75
Dr. T's 4-OP Deluxe	99.75
Dr. T's DX-Heaven	99.75
Dr. T's Kawai K1 Libr.	99.75
Dr. T's KCS 1.6	166.75
Drum Studio	29.75
Dynamic Drums	47.75
Dynamic Studio	129.75
Laser Sound Vol. 1-3	27.50
Midi Recording Studio	44.95
Music Mouse	49.75
Sonix	47.75
Sound Oasis	66.75
Sound Quest Casio CZ	84.75
Sound Quest D-110	99.75
Sound Quest D-50	99.75
Sound Quest DX-7	99.75
Sound Quest DX-7 II	117.75
Sound Quest Generic	84.75
Sound Quest MT-32	99.75
Sound Quest Q-80-ESQ1	99.75
Sound Quest TX-81 C	84.75
Soundscape Pro Midi Studio	124.75
Soundscape Utilities I	39.95
Studio Magic	66.75
Synthia	66.75
Texture	99.75

AntiVirus IV
Sure it will detect a Virus and get it of your System. But more important it allows you to prevent damage by backing up your Boot-tracks ! Background mode included !

Light Pen
Finally there is a Light pen for use with the AMIGA computer made by INKWELL SYSTEMS.
US$ 99.75

BUSINESS

Acquisition	199.75
Analyze 2.0	62.75
BEST Management System	299.75
Critics Choice	159.75
Cygnus ED	69.50
Excellence	189.75
Financial Plus	199.75
Flow	59.95
Gold Spell	29.75
Kindwords	66.75
Logistix	99.75
Maxi Plan 500	99.75
Maxi Plan Plus	129.75
Microfiche Filer	69.75
Microfiche Filer II	112.75
Money Mentor 2.0	59.75
Organize	66.75
Phasar 3.0	66.75
ProScript	32.95
Prowrite 2.0	72.50
Scribble 2.0	57.95
Softwood File II (GS)	79.75
Superbase Personal II	99.75
Superbase Professional	199.75
TexEd	22.75
TexEd Plus	54.75
Word Perfect Library	79.95
Wordperfect	197.75

GRAPHICS / VIDEO

3-Demon	69.75
Aegis Animator	82.50
Aegis Draw 2000	179.75
Analytic Art	39.75
Animate 3D	99.75
Animation Multiplane	56.75
Animator Apprentice	199.75
Butcher 2.0	21.75
Calligrapher new Version 1.05	87.75
Comic Setter	66.75
Deluxe Paint 2.0	94.75
Deluxe Photo Lab	99.75
Deluxe Productions	144.75
Deluxe Video 1.2	94.75
Director	44.95
Dynamic Cad	299.00
Fantavision	39.95
Flipside	39.75
Forms in Flight II	77.75
Gallery 3D	43.75
Interchange	29.75
Intro CAD 2.0	49.75
Kara Fonts	52.75
Lights, Camera, Action	52.75
Modeler 3D	72.50
Pageflipper Plus F/X	99.75
Photon Paint	66.75
Pixmate	49.75
Prism Plus	44.75
Pro Board	379.75
Pro Net	379.75
Pro Video Plus	199.75
Sculpt 3D	69.75
Sculpt 3D (PAL)	69.75
Sculpt 4D	399.75
Silver	99.75
Terrain Disk for Turbo Silver	19.75
TV Show NTSC/PAL	62.95
TV-Text	62.95
Turbo Print	32.75
Turbo Silver	129.75
Video Effects 3D	129.75
Video Titler	99.75
Videoscape 3D	119.75
Window Print II	22.75
X-CAD	359.00
Zoetrope	99.75

GAMES

4th and Inches	29.75
Andromeda Mission	27.75
Amegas	19.75
Arazok's Tomb	24.75
Arkanoid	19.75
Balance of Power	33.75
Barbarian	24.75
Battle Chess	31.75
Black Shadow	25.75
Capone	29.75
Carrier Command	29.95
Centerfold Squares	22.50
Championship Basketball	29.75
Championship Baseball	27.75
Championship Football	29.75
Championship Golf	27.75
Corruption	27.95
Cube Master	19.75
Defender of the Crown	35.75
Destroyer	27.75
Ebonstar	29.75
Emerald Mine	17.75
European Scenery Disk	19.75
Faery Tale Adventure	29.75
Ferrari	32.75
Final Assault	31.75
Fire and Forget	28.75
Flightsimulator	39.75
Foot Man	22.75
Foundation Waste	27.75
Gee Bee Air Rally	27.75
Global Commander	29.95
Grand Slam (Tennis)	29.75
Guild of Thieves	29.75
Hollywood Poker	24.75
Hybris	27.75
Indoor Sports	34.75
Interceptor	34.75
Into the Eagles Nest	24.75
Jet	32.75
Jewels of Darkness	9.75
Karate Kid II	27.75
Knight Orc	29.75
Major Motion	27.75
Menace	21.95
Mindwalker	27.50
Obliterator	29.75
Offshore Warrior	28.75
Pioneer Plague	27.95
Plutos	19.75
Portal	34.75
Ports of Call	29.75
POW	25.75
Q-Ball	19.75
Realm of the Warlock	24.75
Return to Atlantis	34.75
Road Raider	27.75
Roger Rabbit	29.95
Scenery Disk 7/11	19.75
Scenery Disk Europe	19.75
Scenery Disk Japan	19.75
Shadowgate	29.75
Shanghai	27.75
Silent Service	27.75
Silicon Dreams	9.97
Sindbad	33.75
Skychase	27.75
Space Cutter	19.95
Space Quest	32.75
Starglider	29.75
Starglider II	29.75
Starwars	27.75
Street Cat	17.75
Street Sports	14.95
Strip Poker	27.75
Strip Poker Data #4 or #5	14.95
Surgeon	29.75
Terror Pods	29.75
Test Drive	29.75
The 3 Stooges	34.75
The Pawn	29.75
Torch 2081	19.75
Uninvited	33.75
Vampires Empires	29.75
Virus	19.75
Vyper	22.75
Wintergames	14.95
World Class Leaderboard	29.95
World Games	27.75
Zoom	21.75

UTILITIES

Anti-Virus IV	39.95
Arexx	32.75
CLI Mate	27.75
Disk Mechanic	56.75
DOS-2-DOS	42.75
Demonstrator	24.75
Encore	44.95
EZ-Backup	32.75
Facc II	19.75
Fine-Print	32.75
Marauder II	27.75
Project D	32.75
Quarterback	44.95
Raw Copy	39.95

BOOKS

AMIGA Graphic/Sound	17.75
Amiga Dos Manual	19.50
Desktop Video	14.95
Hardware Reference Man.	19.75
Intuition Reference Man.	19.75
ROM-KERNEL: Lib.&Dev.	27.75
ROM-KERNEL: Exec.	19.75

MISCELLANEOUS

A-Talk III	67.95
Intellitype	34.75
Online	44.75
The Works	129.75

New Software in Stock

3-Demon	Final Assault
4th and Inches	Gallery 3D
Andromeda Mission	Menace
Battle Chess	Micro Fiche Filer II
Black Shadow	Modeler 3D
Capone	POW
Comic Setter	ProScript
Comic Setter Funny Figures	Skychase
Comic Setter Science Fiction	Space Cutter
Comic Setter Super Heroes	Starwars
Corruption	Street Cat
Disc Mechanic	Street Sports
Dr. T's Kawai K1 Lib.	Virus
Emerald Mine	World Class Leaderboard
Encore	Zuma Fonts 4

PROGRAMMING-LANGUAGES

A/C Basic Compiler	119.95
A/C Fortran (AB-Soft)	179.75
AMIGA C Compiler	75.00
AMIGA Pascal	77.75
Aztec C Developer	199.75
Aztec C Professional	145.75
Aztec Source Level Debug.	59.75
Benchmark Amiga Library	64.75
Benchmark C-Library	64.75
Benchmark IFF Library	64.75
Benchmark Modula-2	134.75
DSM MC68000 Disassembler	42.75
Lattice-C ++	395.00
Lattice-C Math-Library	69.75
Lattice-C Professional	259.75
Lattice-C Sceen Editor	69.75
Lattice-C Standard	139.75
Lattice-C Text Ed. Pack	49.75
Lattice-C dBC Library	99.75
Lint	69.00
Lisp	99.75
MCC-Pascal	79.95
Macro Assembler f. AMIGA	75.00
Modula-II Commercial	175.00
Modula-II Developer	89.75
Modula-II Standard	59.75
PDO-Pascal	49.75
Power Windows 2.5	56.75
True Basic	69.75

Go ahead !
Call us with your best Quote !
And as allways;
we've got the most 'customer friendly' Terms :
S/H $3.95
30$ Minimum Order
NO Credit Card Fee !
but
In TEXAS add 7% TAX !

All prices are subject to change without notice !

The Future of Amiga Microcomputing

The details Nostradamus never told you

by David Wood

David Wood is a Ph.D. student at Southern Methodist University working on his degree in Electrical Engineering. His area of specialization is image processing. David has been involved with computers since his high school days on an HP 2000 minicomputer; his first microcomputer was an Apple II+.

It's time to pull out the old crystal ball. Unfortunately, mine is as cloudy and cracked as the next person's. As I peer deeply into the ball, I see new and exciting hardware for the Amiga at the start of the next milennium. Is my crystal ball telling the truth or is it leading me by the hand off a cliff? I cannot tell. Let me get some polish and a rag to see what the year 2000 will bring.

Recently, several people have said that the death of the personal computer is near. At first this seems like a ridiculous statement. Personal computers of all shapes and sizes are all around us. They are at work, in the home, in schools - almost everywhere. However, the statement has an element of truth. The machines that are being sold today are much more powerful than the models sold just five years ago. It was not long ago that the largest personal computer had an eight-bit processor, 64K of memory and two floppy disk drives. Standard machines of today have a megabyte of memory, 16 or 32 bit processors, floating point co-processor support, and a hard disk as well as floppy disk drives. Today's machines with their increased power and memory are not merely personal computers but personal workstations.

The future is exciting for the Amiga. The great new features of today will pale in comparison to what my crystal ball shows. Through the ball, I am looking at an ordinary room. At first glance, it looks like an executive's office of today. It has a door, a window overlooking a meadow, a desk and filing cabinet; nothing extraordinary. On the desk is a microcomputer monitor, showing a calendar. A second glance tells another story...

What will today's personal workstations evolve into? Let's take a look around the executive's office. The desk has a perfectly flat, clear surface. Rising out of the surface is a monitor. It looks nothing like the monitors of today. The front is flat and appears to be made of the same material as the surface of the desk. The monitor is only a few inches thick, and the display area is about 12 inches by 12 inches. There is no border, only picture. The calendar image on the screen is in full colour and there are no scan lines. The letters and numbers show no jaggies, even for edges that are close to horizontal or vertical. The resolution must be high.

No keyboard or mouse is seen. Except for the monitor, the desk is empty. Under the surface of the desk, there are five subtly-lit rectangles. The rectangles are different colours and they are labelled. The first says 'Spreadsheet', the second says 'Calendar of Events'. The second rectangle appears to be on the screen at the moment. The rest are labelled as other types of business software.

The running calendar program sounds a tone as the clock turns to 6:00 p.m. The office starts to change. The screen starts to fold back into the desk until the monitor surface is flush with the desk's surface. The window in back of the desk has also changed. The window now appears to look down at least 30 storeys to a busy city street.

The crystal ball fades out of the office and into a biological lab. The room is large and has equipment along all four walls. Off in the corner, there are cages with white mice staring back at me. Nothing seems really out of place in the lab except a large fish tank in the middle of the room. The tank is a cube, at least four feet on each side. Many varieties of exotic fish are swimming around in it. Underneath one edge of the tank is an Amiga. It appears to be running but I cannot tell what it is controlling. There are no keyboards or monitors around (plenty of mice, though).

The crystal ball is starting to fade out again. There are many questions left unanswered. What was the Amiga doing? How did the scientists interface with the Amiga? Too many questions, but no answers.

The crystal ball refocuses on a typical home. From inside the house, I can hear noises from the family in the background. As the swirling mist dissipates, the crystal ball scans the room, I can see the back of an enraptured little boy. He is playing a game and from the sounds I can tell that it is Space Invaders. Some games have enduring appeal and seem never to fade away...

As I get closer, I see that there has been a change made to the game. Now it is in three dimensions: the barns and the descending space creatures have real depth. All of the action is taking place in a glass-like cube, about eight inches on a side. The tank, or holotank, is filled with a not-quite-clear substance that is good enough for home use. Close up, the space creatures show the effects of limited resolution. The game still has the same colours as the (now ancient) 2-D version.

Aha! Something now becomes clear. The Amiga in the biology lab was running a giant holotank. The tank was not a real fish tank at all, but a biological simulation. The Amiga must be handling some experiment.

Eventually the kid gets pounded by the space creatures. He leaves, which lets me approach for a closer look. The label on the bottom edge of the computer says "Amiga 1, The One to Own". Using the magic of the crystal ball, I circle all the way around the machine. The whole machine is simply constructed. There are no cables coming out the front. The left and right side have almost perfectly camouflaged rectangular ports. In fact, there are only a few cables and these come out of the back. One rather thick cable goes to the holotank. Another is the power cord. Two more wires leave the machine. Both are thin wires, one leaving from the right side and the other from the left. There are no other connections on the machine. No keyboard? No mouse?

The crystal ball refocuses on a typical home. As it scans the room, I see the back of a little boy. He is playing a game and from the sounds I can tell that it is Space Invaders...

The kid has left the joystick sitting next to the holotank. It is sitting on the table unconnected. Yet he was using the joystick to control the game. More mysteries.

The crystal ball takes the liberty of searching through the desk drawers. In the top drawer is an assortment of items: a trackball, joydisk and touch tablet. None of these have any cables to attach themselves to the Amiga. In the bottom drawer are a keyboard and a mouse, again with no cables. I cause the crystal ball to bounce off the keyboard's Help key - not an elegant way to use the keyboard, but it does work. The picture on the wall behind the Amiga changes: Mona Lisa disappears and an Amiga menu takes its place.

The menu reads, "Welcome to Your New Amiga. Please select the area that you are interested in and click the left mouse button once. If the mouse is missing, press the corresponding function key." The first entry is an introduction to the hardware options available for all Amigas. I bounce the crystal ball off function key 1.

"Welcome to your new machine", the Amiga responds in an almost human voice. "It has many wonderful capabilities that are awaiting your command. The Amiga is the first truly scalable personal workstation."

The on-screen text is a mixture of juicy details and marketing hype. Skipping the latter, I learn many interesting details about the future...

"Scalability", I read, "is a word often abused by many computer manufacturers. It has become so overused that it almost has no meaning any more, like the phrase 'user-friendly' from the '80s. In the Amiga, scalability is not another buzz-word used to sell computers, but a reality. If two people need to use the computer at once, all you need to do is to open an extra window. No extra hardware to add and no switches to set. The Amiga is completely autoconfiguring. As you will see, scalability applies to just about all the peripherals that you can add to your computer system.

"The Amiga comes standard with a dual channel 16 bit voice/signal digitizer - more than enough resolution for recording stereo. The Amiga supports a sampling rate of up to 1 megahertz per channel. (For top quality audio reproduction, the fastest sampling needed is about 44 kilohertz per channel.) For your signal processing projects, you can handle signals up to 500 kilohertz per channel without distortion from aliasing. Of course, the amount of memory you have determines the length of the sample that you can take.

"Some projects require more signal handling capability. If you need four, eight, or more signals sampled continuously, the Amiga can handle it. With scalability, you can add as many signal processing modules as you need.

"Amiga provides four independent voice/signal output channels. Each channel can be controlled either independently or in concert with others. An almost unlimited number of output channels can be added to the Amiga, depending on your amount of memory and processor power. (More on this later.)

"As is well known, moving your software from one place to another used to be a real problem. Software and data used to be stored in a layer of magnetic particles on thin plastic disks that rotated in devices called disk drives. Unfortunately, the disks were fragile - one day in the summer sun and they would droop. If placed too near a motor or other source of magnetic fields they could lose all their information. They were easily damaged by bending (especially the big 5.25 and 8 inch disks).

"These disks, or 'floppies', are now historical items. The Amiga uses the universally standard software cartridge, which is rectangular in form and about 1 inch by 1 inch by half an inch thick. It is not affected by the environmental conditions that would turn a floppy disk into a plastic potato chip. One end of the cartridge has the contact plates and the other has a coloured dot. Do *not* buy any software if the coloured dot is red.

"To install your program, insert the cartridge in either the left or right hand slots in the Amiga case. Once securely inserted, the dot should be green. Press the green dot. Even the largest programs only take a second or two to load. After loading, the green dot will change to red. The software is now loaded and configured to your system. If for some reason, the software should get damaged or erased from the Amiga, simply reinsert the cartridge and press the red dot. Note: the cartridge cannot be used to load any other system while the dot is red.

"To uninstall and remove any piece of software, insert the correct cartridge and press the red dot. The software will be removed from the machine and copies you have made will no longer run. The cartridge and its contents can now be moved to a new machine.

"In the battle to thwart software pirates, the Amiga uses the recommended National Bureau of Standards (NBS) encryption scheme. All of the central processors have unique two-part serial numbers. One part is public information; the other part is known only to the processor. The software cartridge has its own serial number. It, also, has a public part and a private part. To transfer software from the cartridge, the processor sends an encrypted message using two keys. The processor's private part of the serial number is one key, and the public part of the cartridge's serial number is the other. The cartridge acknowledges the processor by an encrypted response. The cartridge uses the processor's public serial number and its private number as keys. The beauty of this scheme is that both the processor and the cartridge know positively the identity of the other. This locks out software theft by one machine trying to emulate another. Finding the private part to any serial number is a tremendously difficult problem.

The Amiga now supports the 56 kilobit data lines that come standard from the phone company. The 1.544 megabit data lines are also supported. Ask your phone company...

"The engineers of the Amiga realize that a lot of old software is still on magnetic and laser media. As a option, high-density 12-megabyte floppy disk drives can be added to the Amiga. CD WORMs (Compact Disk Write Once Read Many) and WMRMs (Write Many Read Many) are also supported as add-ons. These options are maintained for the convenience of backward compatibility.

"Solid State Cards (SSCs), however, are the mass storage medium of choice for the Amiga. These smart cards are about the size of your credit cards but slightly thicker, and come in many different memory capacities. The Amiga supports up to six SSCs of any capacity at once. The smart cards are inserted in the slot on the front panel of the Amiga. They offer several advantages over older storage technologies: they keep their contents even when there is no power, can handle high and low temperatures and are not affected by magnetic fields. They are small enough to carry in your pocket and strong enough to send through the mail in a business envelope.

"The basic Amiga supports both standard household networks. The first of these is the broadcast network, which works on a very low frequency (VLF) to communicate with your slow-speed peripherals. The keyboards, joysticks, mice, are among the peripherals that talk to the Amiga by this VLF network. The single conductor wire that leaves the back of the Amiga, circles the room and returns to the right side is the antenna. Since the transmitted frequency is so low, there will not be any interference with the neighbors or even the rest of the house. Only the peripherals that are in the antenna loop will be able to talk to the Amiga.

"The second network is the one placed on the house electrical lines. This network is not broadcast over the open air so there is no chance of interference with the radio or television. It is convenient since no cables have to be laid. This network conforms to NBS standards so that it can control all of your network-compatible appliances. Be sure to check your coffee pots, crock pots, lamps, fans, and other appliances for network compatibility so your Amiga can command these. WARNING: Remember to select and set a unique network address for your house. You must do this for the Amiga and all of your network compatible appliances. If you do not do this, you may find yourself controlling someone else's house, or even worse, they could be controlling yours.[1]

"The Amiga comes standard with the high-resolution flat screen. The screen is lightweight and portable. The image is rock stable, without the interlace flicker of yesteryear, and boasts more than 16 million colours displayable. The screen is also much more rugged than in earlier models. The front surface is strong, and resistant to scratching and smudging - now you can pack up your Amiga and screen and take it with you with no fear of damage.

"The newest option to the personal workstation Amiga is the 3-D holotank. Cluttered graphs and charts are much easier to see in 3-D. Games take on a greater realism. CAD has never looked better!"

Reading between the lines, I gather that there is not much software available to support the holotank. I suspect that there will be more as time goes on. According to the marketing hype, 'What's more, the new holotank is surprising affordable'. We will see...

"Printer technology keeps getting better and better. The Amiga comes standard with several serial and parallel ports. Many different printers can be connected to these ports. Many manufacturers make different kinds of dot-matrix, daisy wheel, jet-ink, and laser printers. Choose the one that fits your needs, from economically-priced low-resolution black and white dot-matrix models to the higher-priced high-resolution colour printers.

"Analog telephone modems (300 through 14400 baud) are no longer supported by the Amiga's Operating System. The Amiga now supports the 56 kilobit data lines that come standard from the phone company. The Amiga will also support 1.544 megabit/second data lines as an option. (The 1.544 mbs rate is AT&T's T1 carrier network.[2]) This is a special service that must be contracted from the phone company.

"As an option, Amiga will support one or more high speed professional networks. The network module will support the older 10-megabit Ethernet standard as well as the newer fiber optic network standards. If you are close to the nearest junction box, you can connect directly. You can access computers in the office across town or computers across the country. Screens updates are almost instantaneous, no matter where the originating computer is."

"As you have seen, the Amiga is a remarkably adaptable and expandable computer. This is why we can state that the Amiga is truly a scalable machine. Skeptical? Read on...

"The Amiga comes standard with one main processor (CPU) and one attached processor (APU). The CPU runs the user's programs. The APU (attached processing unit) handles all of the operating system details like task switching, paging, memory allocation, disk access, network control and other jobs. With two processor sets (two each of the CPUs and APUs), the Amiga becomes a true multitasking multiprocessor machine. For even more horsepower, more sets can be added.

As many co-processors as needed can be added to solve the problem. This is the essence of scalability. As your needs grow, your Amiga grows as well...

"Each Amiga comes standard with three co-processors working alongside the main CPU. The first co-processor handles the interfaces to the outside world. It handles the traffic that comes in over the VLF broadcast network as well as the data from the power line network. It is responsible for driving all of the serial and parallel ports (via the VLF network). The fastest devices that this co-processor manages are the WORM and WMRM disks. This input-output processor (IOP) offloads the relatively slow speed I/O responsibilities from the central processor.

"The second co-processor is a graphics handler and bit manipulator. It is responsible for quickly moving blocks of data around in memory. It can also do logic operations on the data during transfer. The graphics and data processor (GDP) can draw basic geometric shapes with one command from the CPU. It draws lines, does hardware fills, changes the colour palette, does automatic page flipping, controls the sprites, displays the fonts, and does other graphical tasks.

"The first two co-processors (IOP and GDP) replace the operation of the three specialized chips - Paula, Agnus and Denise - that gave the original Amigas their power. The third is the math co-processor. It handles regular floating point math operations like addition and multiplication. It also computes some of the more complicated functions like exponentiation, trigonometric and logarithmic operations (transcendental functions). The original Amigas did not have a math co-processor. All floating point operations were handled in software, which made the early Amigas cheaper to manufacture, but slowed down math-intensive programs.

"Depending on the application, extra processors and co-processors can be added to increase the power of the machine without sacrificing flexibility and compatibility. If the application needs more CPU resources, then another CPU and APU set can be added. If more floating point power is needed, then the math co-processor can be added to the CPU and APU set. For more graphics horsepower, add more GDPs. If more external I/O lines are needed, then more IOPs can be added. As many co-processors as needed can be added to solve the problem. This is the essence of scalability. In all, the basic Amiga offers so much. As your needs grow, your Amiga grows as well. This prevents early obsolescence of your investment. Press any key to return the Mona Lisa".

Slowly the crystal ball fades out. It has given us an interesting look into the world of the future. Only time will tell what will come to pass.

Annotated Bibliography

[1] You can control your household appliances today using the BSR X-10 system. A control console sends signals over the electrical lines to modules that plug into the wall sockets. The wall modules control your lamps and appliances. The lamp modules gives 0% to 100% brightness control as well as remote on and off. Automatic timers, remote control through telephone lines, hand held remotes, as well as controllers driven by RS-232 are available. More information about BSR is available from Radio Shack or DAK Industries (8200 Remmet Ave., Canoga Park, CA 91304).

[2] In the AT&T telephone network, the T1 carrier is a 1.544 megabits per second digital channel. The T1 was designed in the 1960s to operate over twisted wire pair cables and carry 24 voice channels. There are many books on telecommunications of varying difficulties. The source that I used was "Future Developments in Telecommunications, second edition" by James Martin. The book is printed in Englewood Cliffs, New Jersey by Prentice-Hall, Inc. The book notes that "[the Bell T1 carrier] is likely to be a standard for decades to come" (page 63).

Acknowledgements

I would like to thank Bill Knight, Ed Corr and Dr. Samir Soliman for their technical help. I would like to thank Betty Clay for the time and effort she put into proofreading and criticising this article.

Creating Run-Time Libraries

Adding your own functions to the system

by Steve Simpson, B.Sc.(Hons.), M.Sc.
Copyright © 1988 S. Simpson

Steve Simpson is a freelance consultant programmer of the Amiga and IBM PC. He is an ex-astronomer, has lived in Sweden for five years and translates technical documents. His current interests include systems, DOS utilities development and industrial process control. He is currently engaged in SideCar routines development. He is the author of DiskRepair, a disk editing/repair utility for the Amiga, which is available from him.

Anyone who has programmed the Amiga will be familiar with the use of run-time, or "shared", libraries. These are either contained within the main Kickstart ROM (the most familiar being the *graphics.library*, *exec.library* and *dos.library* which contain the graphics, Exec kernal and AmigaDOS routines), or they are files with the *.library* suffix in the *libs* directory on the system disk. The run-time library lies at the heart of the Amiga's soft architecture.

As part of a larger project, I wanted to build a library in which to maintain my most commonly used routines. These could then be accessed from my programs in the Amiga multitasking system, rather than keeping separate copies of the routines in execution modules. This article gives a detailed description of how a run-time library can be constructed and added to the Amiga system.

What is a run-time library?

In a soft machine architecture, system routines and jump tables do not reside at absolute addresses. They are loaded into memory when they are needed, wherever there is room for them. This results in a flexible system with efficient use of memory, which is particularly important in a multitasking system. Since there aren't any fixed addresses, libraries are easily upgraded.

Note that, in the Amiga programming environment, the word 'library' is used in two different contexts: (1) a collection of run-time routines in Kickstart memory or loaded from disk and accessed via jump tables in RAM; (2) standard definitions and routines used at link time which also contain interface routines to run-time library routines.

Figure 1 illustrates the block structure of a library, which comprises the library structure and data areas, together with a table of jump vectors (usually six bytes wide) for the library routines. These library routines do not appear with the main library structure, but are loaded wherever there is space.

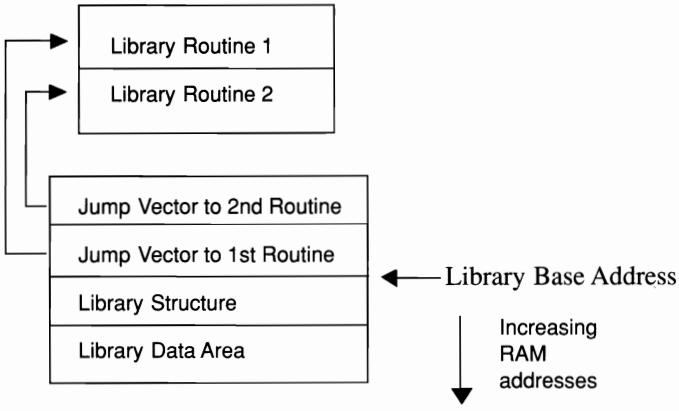


Figure 1: *The relationship between the components of a run-time library and library routines.*

The OpenLibrary() function scans a list of library names maintained by Exec, and from this list it determines the library's base address. This address is returned in register D0 and the library data area extends to positive displacements from it.

The library node or structure is defined in the C header file *exec/libraries.h* as follows:

```
struct Library      {
    struct  Node lib_Node;
    UBYTE   lib_Flags;
    UBYTE   lib_pad;
```

```
UWORD    lib_NegSize;
UWORD    lib_PosSize;
UWORD    lib_Version;
UWORD    lib_Revision;
APTR     lib_IdString;
ULONG    lib_Sum;
UWORD    lib_OpenCnt;
};
```

The individual elements of the Library structure are as follows:

lib_Node: a Node substructure. Within this, the ln_Name parameter points to the name of the library; ln_Pri determines where in the library list the library sits; ln_Type is NT_LIBRARY to show that the node is a library node.

lib_Flags: library flags: LIBF_SUMMING indicates that a task is currently running a checksum on the library; LIBF_CHANGED, one or more entries in the vector table have been changed; LIBF_SUMUSED, the library designer wants a checksum fault to cause a system panic (crash!).

lib_NegSize: number of bytes preceding the library structure itself. This yields the number of entries in the vector table.

lib_PosSize: number of bytes following the library structure. This is the data area.

lib_Version: version number of the library.

lib_Revision: revision number of the library.

lib_IdString: pointer to NULL-terminated text string which identifies the library.

lib_Sum: current value of the library's checksum.

lib_OpenCnt: incremented each time the library is opened, and decremented each time it is closed.

As an example, we can look at the *graphics.library*. Its jump table starts in the following way:

```
offsets

FFCA (-60) JMP $00FC7FF8 Text
FFCA (-54) JMP $00FC7F98 TextLength
FFCA (-48) JMP $00FC7CC0 ClearScreen
FFCA (-42) JMP $00FC7C88 ClearEOL
FFCA (-36) JMP $00FC7D24 BltTemplate
FFCA (-30) JMP $00FC746A BltBitMap
FFCA (-24) JMP $00FC6CD4 ExtFunc
FFCA (-18) JMP $00FC6CD8 Expunge
FFCA (-12) JMP $00FC6CD8 Close
FFCA (-6)  JMP $00FC6CD4 Open
0          Library Base Address
```

The jump destinations are not always the same - they depend on where AmigaDOS has loaded the library into RAM. However, the order of the entries in the jump table *does* remain constant. The jump table entries build downwards in memory from the library base address; that is, towards negative offsets. Thus, by knowing the library base address for *graphics.library* (called GfxBase), we can get to any jump table entry by indirect addressing. If register A6 contains GfxBase, then the assembler command:

```
JSR -48(A6)
```

will call the *ClearScreen* routine. In assembler, the offset is either declared as global in the assembler source file and resolved during linking or is defined as a constant in an appropriate *.i* file. A globally-defined offset would be prefixed with _LVO and defined to be -48 ($FFD0), for example:

```
JSR _LVOClearScreen(A6)
```

The offset constant in a *.i* file would appear as an offset beyond the standard library routines offsets, thus:

```
LIBINIT
    .....
    LIBDEF CLEAREOL
    LIBDEF CLEARSCREEN
    LIBDEF TEXTLENGTH
    .....
```

Where LIBINIT defines the standard routine offsets (Open, Close, Expunge and ExtFunc) and the following would be used in the assembler source code to jump to the ClearScreen routine:

```
LINKLIB CLEARSCREEN, A6
```

LINKLIB is a macro that will expand as follows:

```
MOVE.L  A6.-(SP)
MOVE.L  A2,A6
JSR    -48(A6)
MOVE.L (SP)+,A6
```

The listings illustrate how to use both calling systems and how to implement user-defined run-time library routines.

Building a run-time library

The *ROM Kernal Manual: Libraries and Devices* (RKM) gives the code for an example skeleton library. However, it does not describe how to implement it and make it available to the Amiga system. Additionally, there are also a number of severe errors in the RKM listing making it almost useless, as I found out to my cost!

Listing 1 gives the code for *mylib.library*. The library code is supported by *libsupp.i* and *mylib_def.i* include modules (Listings 2 and 3 respectively).

The first instruction in the library sets the D0 register to zero and returns immediately. This is a safeguard just in case someone tries to execute the library.

The ROM-tag structure gives the system information about the location of set-up routines and data. Its format as a C structure (found in the C header file *exec/resident.h*) would be:

```
struct Resident  {
    UWORD    rt_MatchWord;
    struct   Resident *rt_MatchTag;
    APTR     rt_EndSkip;
    UBYTE    rt_Flags;
    UBYTE    rt_Version;
    UBYTE    rt_Type;
    BYTE     rt_Pri;
    char     *rt_Name;
    char     *rt_IdString;
    APTR     rt_Init;
    };
```

rt_MatchWord: RTC_MATCHWORD (used to identify the ROM-tag structure).

rt_MatchTag: pointer to the next ROM-tag structure.

rt_EndSkip: address of the library end code.

rt_Flags: ROM-tag flags: either RTF_AUTOINIT or RTF_COLDSTART.

rt_Version: version number.

rt_Type: ROM-tag type: either NT_LIBRARY, NT_DEVICE or NT_RESOURCE.

rt_Pri: the library priority - usually zero. This gives the preferred location in the Exec library list.

rt_Name: pointer to a NULL-terminated name string. In this example library "mylib.library".

rt_IdString: pointer to a NULL-terminated string. This is the library's identifier tag and is used in library support and maintenance. It helps to identify different versions of a library, and its format is:

name version.revision (dd mmm yyyy),CR,LF,NULL

e.g. "mylib 1.0 (06 May 1988) \015\012\0"

(\015 and \012 are octal equivalents of CR (carriage return) and LF (line feed).)

rt_Init: pointer to the initialisation table for this library. Points to a table of addresses for initialisation of data and any initialisation routines. This table would have the following C structure:

```
struct Init  {
    ULONG space;
    ULONG funcTable;
    ULONG dataTable;
    ULONG initRoutine;
    };
```

space - the total data space size in bytes required by the library. This is the size of the library structure plus user library data space.

funcTable - pointer to table of function initializers or addresses to the library's functions (routines).

dataTable - pointer to table of data initializers.

initRoutine - pointer to the routine to initialize the library.

These parameters are used as input to the MakeLibrary() library initialisation routine which is discussed later.

In Listing 1, the *funcTable:* code references the absolute addresses of the routines in the library. These are the routines that will be allocated places in the jump table. These jump table entries are usually six bytes wide, but four- or two-byte wide entries can be used depending on how close in memory the library routines are to the library structure. In the case of four-byte wide jump table entries, the routines must reside within 32K of the library structure. The table is terminated with -1L.

If the first long word of the table is -1, then the table entries are relative word displacements. The table is terminated with -1L.

The code for *dataTable:* in Listing 1 initializes the static data structures. In this example we initialize the library structure with various values, but you could use this table to initialize your own data areas after allocation with AllocMem(). The format used is the same as that required for input to the InitStruct() routine. This routine clears a memory area, preserving those words whose data and offset values are given in the initialisation table. The macros used for dataTable in Listing 1 can be ilustrated by their equivalent C structures:

```
struct InitByte  {
    UBYTE ib_command;
    UBYTE ib_count;
    UWORD ib_offset;
    BYTE  ib_value;
    UBYTE ib_pad;
    };

struct InitWord  {
    UBYTE iw_command;
    UBYTE iw_count;
    UWORD iw_offset;
    UBYTE iw_value;
    };
```

```
struct InitLong  {
UBYTE il_command;
UBYTE il_count;
UWORD il_offset;
LONG  il_value;
};
```

In the above, the command format is **ddssnnnn**, where:

dd is the destination type (and size):
00 - next is destination, nnnn is count
01 - next is destination, nnnn is repeat
10 - destination offset is next byte, nnnn is count
11 - destination offset is next rptr, nnnn is count

ss is the size and location of the source:
00 - long, from next two aligned words
01 - word, form next aligned word
10 - byte, from next byte
11 - ERROR - will cause ALERT

nnnn is the count or repeat:
count - the (number+1) of source items to copy
repeat - the source is copied (number+1) times

The offset is the offset into the data area to be initialized; in Listing 1, we are initialising various parameters in the library structure. The dataTable sequence is terminated with 0L.

After MakeLibrary() has allocated the library, initRoutine is called. the user can include custom library specific initialisations in this routine. In the example, Listing 1, copies of the system base pointer, the library segment pointer and the DOS library base pointer saved. The DOS library is also opened, otherwise an alert is shown on error.

System interface

A library must have a minimum of four standard functions: Open, Close, Expunge and ExtFunc. When the user calls OpenLibrary(), CloseLibrary() or RemoveLibrary(), they will eventually be translated into Open, Close or Expunge. ExtFunc is reserved.

User library routines are positioned after these functions. If a routine is developed outside the main library source code, then it must be declared as global (using XREF <function name> in assembler). User routines can be developed in assembler or C (or any other language). These routines are linked to the main library code during the linking process.

Interface for calling library routines

Amiga convention uses the A6 register for the base address of the library being called. Registers D0, D1, A0 and A1 are scratch registers and are used to pass values or pointers to the library routines. D0 is used to return a value from a library routine.

This system is illustrated in the example interface routines in Listings 4, 5 and 6. MyFunc0, for example, may be called from a C program by including MyFunc0() in the source code.

Parameters could be passed to the library routines by putting their values in registers, which is fine for calls made from assembler. However, C places function call parameters on the stack and Listing 6 shows how these values are extracted and placed in registers, before jumping to the called library routine. This represents a source of inefficiency when programming in C.

Routines must also reference the user's own library base pointer (eg. *_MyBase* in the example listings).

Interface routines could be kept in a scanned library and called during linking. Example routine interfaces (all those with _ prefix) could be kept in the linker scanned object module library *my.lib*.

Library routines written in assembler can now make use of register values directly. But what if a routine is developed in C? Listing 7 illustrates such a routine; after entering the routine, the value placed into A0 after extraction from the stack by the interface routine MyFunc2, is put back onto the stack. This is implemented as an in-line assembler command

```
MOVE.L A0,-4(A5)
```

where A5 holds the current stack pointer. In this example, -4(A5) refers to the local variable VAL in the routine __MyFunc2. This convention is compiler-dependent - and you will need to check your compiler's documentation.

We can then perform the example calculation and return the answer in D0. (Note: we use __MyFunc2 to differentiate from _MyFunc2 - the former is the library routine in C, the latter is the C calling interface to __MyFunc2.)

Using the run-time library

The user-defined run-time library is opened in the usual way, by a call to:

```
libBase = OpenLibrary(name, version);
```

where: `struct Library *libBase;`

The version number allows the use of different versions of a library. Listing 9 illustrates how the library is opened and closed. It also shows how to call the user-implemented library routines, MyFunc0(), MyFunc1() and MyFunc2().

After finishing with the library it is closed using:

```
CloseLibrary(libBase);
```

Installing a library in the system

The component parts of the user run-time library are assembled or compiled with the option to omit the *.begin* (for Manx users) or *startup.obj* (for Lattice users) routine. We don't need the load code since we are effectively providing our own. All components are then linked along with the standard Amiga libraries. The resultant file is the run-time library file, *my.library* in the examples given.

Listing 8 illustrates how a user-defined run-time library is added to the system. The library is loaded into memory by LoadSeg(). A call to MakeLibrary() initializes the library's jump tables, sets up data areas and performs user initialisation. Parameters to MakeLibrary() include a pointer to the table of library function addresses, a table of data initialisations, a pointer to the user library initialisation routine, the space required by the library in bytes, and a BCPL pointer to the segment list returned from LoadSeg(). The space required parameter is the size of the library structure plus and user data area. The call to MakeLibrary() looks like this:

```
libBase = MakeLibrary (funcInit,
strucInit, libInit, dataSize, segList);
```

where: `struct Library *libBase;`

If structInit is NULL, the InitStruct() function is not called; the library either has no data or the data area is specified in some other way. If libInit is NULL the Init routine is not called, and the library is not initialized in the usual way.

MakeLibrary() returns a pointer to the library base (libBase). This pointer is used as input to AddLibrary(). which adds the library name to the system library list, thus making the library available to every task. The library is not opened by AddLibrary().

There are, in addition, two other ways of making a run-time library available to the system. These methods are probably easier, but for them to work, you must follow exactly the library format given here in the assembler examples.

On issuing an OpenLibrary() request, the system will search the list of available and installed libraries. If a match is not found, the system looks for the specified library in the directory specified by the AmigaDOS logical device LIBS:. If found, the library is loaded, initialized and opened and a pointer to the library base is returned.

Alternatively, a library may be saved in the "Expansion" directory of the system disk. (Useful if you have, for example, a SideCar/BridgeBoard.) A Workbench icon *.info* file must be created for the library, and should be named *<library_name>.info*. On boot-up, the routines contained in the *expansion.library* will look in the "Expansion" directory, loading and initializing any libraries with the *.info* suffix.

Debugging libraries

This section could, in itself, form the subject of a complete article. We can't hope to cover more than just the very bones; most of the discussion comes from first-hand knowledge of trying to fathom out such questions as "Just where did that pointer go to?!".

Firstly, a word about parameter passing from C routines to assembler: You should determine the parameter passing convention used by your compiler. The ability to enter assembler code directly into the C source code is a distinct advantage.

If possible, try to debug all routines external to the library under development. This will probably entail opening an incomplete library in order to make calls to existing routines within it. It helps to have a good debugger, especially one that will allow source code debugging.

Ensure that the order you assign for your routines in the library function table matches that order specified in the assembler header file defining LIBINIT (in the example: *mylib_def.i*).

Summary

The concept of the run-time library increases the efficiency of the Amiga system, allowing programs to use a standard set of routines. A user can keep most commonly used routines in a run-time library. This article should clarify an area of the Amiga environment that is not well documented in the literature. Devices and resources are closely related to libraries and this article has served to introduce their implementation by showing how a library is created.

The example listings were developed using the Manx Aztec C68K v3.60a C compiler and assembler, and debugged using the Manx SDB source level debugger.

• • • • •

Further reading

ROM Kernel Reference Manual: Libraries and Devices, Addison-Wesley

ROM Kernel Reference Manual: Exec, Addison-Wesley

The Kickstart Guide to the Amiga, Ariadne Software Ltd.

Commodore Amiga Inc., *The AmigaDOS Manual*, 2nd Edition, Bantam

Mortimore, E., *Amiga Programmers Guide, Vol. 1*, 2nd Edition, Sybex

Williams, S., *Programming the 68000*, Sybex

Listing 1:

```
;-----------------------------------------------------------------------
;
; mylib.asm -- example library code
;
; Example of how a user system run-time library is constructed.
;
; History:
;    88-05-06  created by Steve Simpson
;                         Skarpskyttev0C
;                         S-222 42 Lund
;                         Sweden
;
; assemble:   as mylib.asm
; link:       ln -o mylib.library mylib.o _myfunc2.o c.lib
;
;-----------------------------------------------------------------------

 SECTION     section

 NOLIST
 INCLUDE "exec/types.i"
 INCLUDE "exec/nodes.i"
 INCLUDE "exec/lists.i"
 INCLUDE "exec/libraries.i"
 INCLUDE "exec/alerts.i"
 INCLUDE "exec/resident.i"
 INCLUDE "exec/initializers.i"
 INCLUDE "libraries/dos.i"
 INCLUDE "libsupp.i"
 INCLUDE "mylib_def.i"
 LIST

;-----------------------------------------------------------------------
; external definitions and references
;-----------------------------------------------------------------------
 XDEF     Init            ;library initialisation
 XDEF     Null
 XDEF     myName
 XDEF     Open            ;library standard routines
 XDEF     Close           ;   --"--
 XDEF     Expunge         ;   --"--
 XDEF     ExtFunc         ;   --"--
 XDEF     MyFunc0         ;library user routines
 XDEF     MyFunc1         ;   --"--
 XDEF     __MyFunc2       ;   --"--  (this one in C)

 XREF     _AbsExecBase    ;the only fixed address in Amiga system

 XLIB     OpenLibrary
 XLIB     CloseLibrary
 XLIB     Alert
 XLIB     FreeMem
 XLIB     Remove

;-----------------------------------------------------------------------
;The first executable location.  This returns an error in case someone
;tried to run the library as a program.
;-----------------------------------------------------------------------
LibStart:
        CLEAR    d0
        rts

;-----------------------------------------------------------------------
; various definitions and equates
;-----------------------------------------------------------------------
MYPRI    EQU      0   ;lib priority - position in system library list
VERSION  EQU      1   ;major version number
REVISION EQU      1   ;particular revision.  This should uniquely
                      ;identify the library

;-----------------------------------------------------------------------
;A romtag structure.
;exec and ramlib look for this structure during the loading operation
;to discover such as where to start running it, for example.
;In C this would be equivalent to the Resident structure
;-----------------------------------------------------------------------
initLibDescrip:                 ;structure RT,0
        dc.w     RTC_MATCHWORD        ;UWORD RT_MATCHWORD
        dc.l     initLibDescrip       ;APTR  RT_MATCHTAG
        dc.l     EndLibCode           ;APTR  RT_ENDSKIP
        dc.b     RTF_AUTOINIT         ;UBYTE RT_FLAGS
        dc.b     VERSION              ;UBYTE RT_VERSION
        dc.b     NT_LIBRARY           ;UBYTE RT_TYPE
        dc.b     MYPRI                ;UBYTE RT_PRI
        dc.l     myName               ;APTR  RT_NAME
        dc.l     libIdString          ;APTR  RT_IDSTRING
        dc.l     libInit              ;APRT  RT_INIT
                                      ;LABEL RT_SIZE

myName         MYLIBNAME              ;my library name

;-----------------------------------------------------------------------
;library identifier tag used in supporting the library
;format is 'name version.revision (dd mon yyyy)',<cr>,<lf>,<null>
;-----------------------------------------------------------------------
libIdString    dc.b     'mylib 1.0 (06 May 1988)',13,10,0

dosName        DOSNAME                ;name of dos library to open

        ds.w     0                    ;we need word alignmemt

;-----------------------------------------------------------------------
;the romtag specifies "RTF_AUTOINIT".  The RT_INIT structure field
;points to the following table.  If RTF_AUTOINIT was not set, then
;RT_INIT would point to a initialisation routine to run.
;These data are used by the loading program as parameters to
;AddLibrary() library initialisation.
;-----------------------------------------------------------------------
libInit:
        dc.l     MyLib_Sizeof         ;data space size
        dc.l     funcTable            ;pointer to function initialisers
        dc.l     dataTable            ;pointer to data initialisers
        dc.l     initRoutine          ;routine to run

;-----------------------------------------------------------------------
;table of addresses of mylib functions
;-----------------------------------------------------------------------
funcTable:
    ;standard system routines - these MUST always be included
        dc.l     Open
        dc.l     Close
        dc.l     Expunge
        dc.l     ExtFunc
    ;my library's definitions
        dc.l     MyFunc0
        dc.l     MyFunc1
        dc.l     __MyFunc2
    ;function table end marker
        dc.l     -1

;-----------------------------------------------------------------------
;this data table initialises static data structures.  The format is
;specified in exec/InitStruct routines manual pages.
;The first argument is the offset from the library base for this
;byte/word/long.  The second argument is the value to put in that cell.
;The table is null terminated.
;-----------------------------------------------------------------------
dataTable:
        INITBYTE    LH_TYPE,NT_LIBRARY
        INITLONG    LN_NAME,myName
        INITBYTE    LIB_FLAGS,LIBF_SUMUSED!LIBF_CHANGED
        INITWORD    LIB_VERSION,VERSION
```

```
        INITWORD    LIB_REVISION,REVISION
        INITLONG    LIB_IDSTRING,idString
        dc.l        0

;-----------------------------------------------------------------------
;This is the initialisation routine - it is called after the library
;has been allocated.
;If it returns non-zero then the library will be linked into the
;library list at position given by priority.
;-----------------------------------------------------------------------
initRoutine:        ;( D0: lib. ptr., A0:segment list)
    ;get the library pointer into a A5
        move.l      a5,-(sp)
        move.l      d0,a5

    ;save pointer to exec
        move.l      a6,ml_SysLib(a5)

    ;save a pointer to loaded code segment list
        move.l      a0,ml_SegList(a5)

    ;open the dos.library
        lea         dosName(pc),a1
        CLEAR       d0
        CALLSYS     OpenLibrary

        move.l      d0,ml_DosLib(a5)
        bne.s       1$

    ;cant open dos
        ALERT       AG_OpenLib!AO_DOSLib

1$:
    ;build the static data that we need
    ;
    ;library specific initialisations go here
    ;

        move.l      a5,d0
        move.l      (sp)+,a5
        rts

;-----------------------------------------------------------------------
;System interface commands begin here.
;Calling OpenLibrary/CloseLibrary/RemoveLibrary translates into a call
;to the routines Open/Close/Expunge.
;The library base pointer is in A6.  Task switching has been turned off
;while in these routines (via Forbid/Permit), so don't hang about!
;-----------------------------------------------------------------------

;-----------------------------------------------------------------------
;Open returns the library pointer in D0 if the open was successful.
;If the open failed then null is returned.
;-----------------------------------------------------------------------
Open:     ;(libptr: A6,  version: D0
    ;increment library open counter
        addq.w      #1,LIB_OPENCNT(a6)

    ;prevent delayed expunges
        bclr        #LIBB_DELEXP,ml_Flags(a6)

        move.l      a6,d0
        rts

;-----------------------------------------------------------------------
;If the library is no longer open, Close returns the segment list if
;the delayed expunge flag is set.  Otherwise Close return NULL.
;-----------------------------------------------------------------------
Close:    ;(libptr: a6)
    ;set the return value
        CLEAR       d0

    ;decrement library open counter
        subq.w      #1,LIB_OPENCNT(a6)

    ;anyone got us open?
        bne.s       1$

    ;any delayed expunge pending?
        btst        #LIBB_DELEXP,ml_Flags(a6)
        beq.s       1$

    ;do the expunge
        bsr         Expunge

1$:
        rts

;-----------------------------------------------------------------------
;If the library is no longer open, then Expunge returns the segment
;list for the library code.  Otherwise the delayed expunge flag is
;set and NULL is returned in D0.
;Expunge should never Wait, or take a long time to complete.
;-----------------------------------------------------------------------
Expunge:     ;(libptr: A6)
        movem.l     d2/a5/a6,-(sp)
        move.l      a6,a5
        move.l      ml_SysLib(a5),a6

    ;anyone got us open?
        tst.w       LIB_OPENCNT(a5)
        beq         1$

    ;if still open - set delayed expunge flag
        bset        #LIBB_DELEXP,ml_Flags(a5)
        CLEAR       d0
        bra.s       Expunge_End

1$:
    ;we can remove ourselves here
        move.l      ml_SegList(a5),d2

    ;unlink from library list
        move.l      a5,a1
        CALLSYS     Remove

    ;============================
    ;library specific closings here
    ;============================

    ;close the dos library
        move.l      ml_DosLib(a5),a1
        CALLSYS     CloseLibrary

    ;free our memory
        CLEAR       d0
        move.l      a5,a1
        move.w      LIB_NEGSIZE(a5),d0

        sub.l       d0,a1
        add.w       LIB_POSSIZE(a5),d0

        CALLSYS     FreeMem

    ;setup our return value - the seglist
        move.l      d2,d0

Expunge_End:
        movem.l     (sp)+,d2/a5/a6
        rts

;-----------------------------------------------------------------------
;ExtFunc just returns 0 in D0 register.
;-----------------------------------------------------------------------
```

```
ExtFunc:
        CLEAR       d0
        rts

;-----------------------------------------------------------------------
;The library specific commands begin here
;In this example library there are 3 library specific routines -
;the third __MyFunc2 is written in C, and is included in the link
;list for the library.
;-----------------------------------------------------------------------

;-----------------------------------------------------------------------
; MyFunc0 - just returns value 0 in the D0 register
;-----------------------------------------------------------------------
MyFunc0:
        CLEAR       d0
        rts

;-----------------------------------------------------------------------
; MyFunc1 - returns contents of register A6 in the D0 register
;-----------------------------------------------------------------------
MyFunc1:
        move.l      a6,d0
        rts

;-----------------------------------------------------------------------
;This is the end of our library code
;-----------------------------------------------------------------------
EndLibCode:
        END
```

Listing 2:

```
;-----------------------------------------------------------------------
; libsupp.i
;
; supplemental macros
;
; History:
;    88-05-06  created
;-----------------------------------------------------------------------

CLEAR           MACRO
    moveq    #0,\1
    ENDM

LINKSYS         MACRO
    LINKLIB  _LVO\1,\2
    ENDM

CALLSYS         MACRO
    CALLLIB  _LVO\1
    ENDM

XLIB            MACRO
    XREF     _LVO\1
    ENDM
```

Listing 3:

```
;-----------------------------------------------------------------
; mylib_def.i
;
; assembler defines for mylib.library
;
; History:
;    88-05-06  created
;-----------------------------------------------------------------

;--library function offsets from base structure
LIBINIT
 LIBDEF     MYFUNC0
 LIBDEF     MYFUNC1
 LIBDEF     MYFUNC2

;--mylib data area structure - this is appended to Library structure
;--at offset LIB_SIZE
 STRUCTURE   MyLib,LIB_SIZE
   ULONG  ml_SysLib
   ULONG  ml_DosLib
   ULONG  ml_SegList
   UBYTE  ml_Flags
   UBYTE  ml_pad
   LABEL  MyLib_Sizeof

;--name of my library
MYLIBNAME          MACRO
      dc.b     'mylib.library',0
      ENDM
```

Listing 4:

```
;-----------------------------------------------------------------------
;  myfunc0.asm
;
; Interface between calling program and jump table for mylib.library
; routine MyFunc0
;
; History:
;    88-05-06  created
;-----------------------------------------------------------------------

 NOLIST
 INCLUDE "exec/types.i"
 INCLUDE "exec/libraries.i"
 INCLUDE "libsupp.i"
 INCLUDE "mylib_def.i"
 LIST

 XREF  _MyBase
 XREF  _MyFunc0

;-----------------------------------------------------------------------
; C calling sequence:  ret = MyFunc0();
;          registers:   d0
;-----------------------------------------------------------------------
_MyFunc0:
    ;--put library base into A2 register
        move.l   _MyBase,a2

    ;--make routine call
 LINKLIB MYFUNC0,a2

        rts

        END
```

Listing 5:

```
;-----------------------------------------------------------------------
;  myfunc1.asm
;
; Interface between calling program and jump table in mylib.library
; for routine MyFunc1
```

```
;
; History:
;    88-05-06  created
;-----------------------------------------------------------------------

 NOLIST
 INCLUDE "exec/types.i"
 INCLUDE "exec/libraries.i"
 INCLUDE "libsupp.i"
 INCLUDE "mylib_def.i"
 LIST

 XREF  _MyBase
 XREF  _MyFunc1

;-----------------------------------------------------------------------
; C calling sequence:   ret = MyFunc1();
;          registers:    d0
;-----------------------------------------------------------------------
_MyFunc1:
    ;--put library base into A2 register
        move.l    _MyBase,a2

    ;--jump to library table
 LINKLIB MYFUNC1,a2

      rts

      END
```

Listing 6:

```
;
;  myfunc2.asm
;
; Interface between calling program and jump table in mylib.library
; for routine MyFunc2
;
; History:
;    88-05-06  created

 NOLIST
 INCLUDE "exec/types.i"
 INCLUDE "exec/libraries.i"
 INCLUDE "libsupp.i"
 INCLUDE "mylib_def.i"
 LIST

 XREF  _MyBase
 XREF  _MyFunc2

;-----------------------------------------------------------
; C calling sequence:  ret = MyFunc2(val);
;          registers:   d0            a0
;-----------------------------------------------------------
_MyFunc2:

    ;--get input value from stack into register
        move.l   4(sp),a0

    ;--put base into register
        move.l    _MyBase,a2

    ;--now jump to the library jump table
 LINKLIB MYFUNC2,a2

        rts

        END
```

Listing 7:

```
/*****************************************************************
 * _myfunc2.c
 *
 * C routine (__MyFunc2) installed in library.
 * Since when making a call to a library routine in the assembler
 * interface data are put into registers, then we must recover the
 * data and put them into local variables in any C routine.
 *
 * History:
 *  88-05-06  created
 *
 * compile:  cc +b _myfunc2
 *    we don't need the startup code for this module
 *****************************************************************/

#include <exec/types.h>

/*************************************************************
 *
 * Assembler call:   input register: a0
 *
 *************************************************************/
ULONG _MyFunc2()
{
  ULONG val;

  /* assembler directives */
#asm
  ;--This is my library interface - it gets the value put into register
  ;--A0 by the routine MyFunc2.
  ;--The value is then put into the local variable before being used
  ;--by this C routine
  ;--Register A5 is the local stack pointer, the compiler allocating
  ;--space on the program stack for this routines local variables.
      move.l   a0,-4(a5)
#endasm

    /* do the example calculation */
  val = val * 2;

    /* return the value - this goes into d0 */
  return(val);

} /* end _MyFunc2 */
```

Listing.8:

```
/*****************************************************************
 * add_lib.c
 *
 * Module to add a user library to the library list maintained
 * by the system.  After adding the user library is available to
 * all subsequent programs
 *
 * History:
 *   88-05-06  created
 *
 * compile:  cc add_lib
 * link:     ln add_lib -lc
 *****************************************************************/
#include <functions.h>
#include <exec/types.h>
#include <exec/libraries.h>
```

```
#include <exec/resident.h>
#include <libraries/dos.h>
#include <libraries/dosextens.h>
#include <mylib.h>

ULONG segList,      /* BCPL seglist pointer */
      codeloc;      /* pointer to code location */

struct Init {
  ULONG space;        /* data space size */
  ULONG funcTable;    /* ptr.to func offsets table */
  ULONG dataTable;    /* ptr.to initStruct table */
  ULONG initRoutine;  /* ptr.to library initRoutine */
} *init;

struct Resident *libRes;  /* ptr.to library rom tag area */

ULONG space, funcTable, dataTable, initRoutine;

struct MyLibrary *mylib;


main()
{

      /* load the library segment */
  segList = (ULONG)LoadSeg("mylib.library");
  if (segList==NULL) {
    printf("Can't load mylib.library\n");
    exit(0);
  }

      /* address of start of library code in memory
       * multiply by 4 since segList is a BCPL pointer */
  codeloc = segList * 4;

      /* address of library resident rom tag data area */
  libRes = (struct Resident *)(codeloc + 8);

      /* address to initialisation data table */
  init = (struct Init *)libRes->rt_Init;

      /* extract data from initialisation table */
  space = init->space;
  funcTable = init->funcTable;
  dataTable = init->dataTable;
  initRoutine = init->initRoutine;

      /* initialise the library */
  mylib = (struct MyLibrary *)MakeLibrary(funcTable, dataTable,
                                          initRoutine, space, segList);

      /* make the library known to the system */
  AddLibrary(mylib);

  printf("mylib.library added to system\n");

}
```

Listing 9:

```
/*****************************************************************
 * my_main.c
 *
 * Test program for mylib.library
 *
 * History:
 *   88-05-06  created by Steve Simpson
 *
 * compile:  cc my_main
 * link:     ln my_main -lmylib -lc
 *****************************************************************/

#include <functions.h>
#include <exec/types.h>
#include <exec/libraries.h>
#include <mylib.h>

  /* my defined library base - this global and is accessible to the
   * assembler routines */
struct MyLibrary *MyBase;

main()
{
  ULONG ret, val;

    /* open my library */
  MyBase = (struct MyLibrary *)OpenLibrary("mylib.library",1L);
  if (MyBase==NULL) {
    printf("Can't open mylib.library\n");
    exit(0);
  }

    /* func0 resets the D0 register and returns its value */
  ret = MyFunc0();
  printf("ret(0)=%lx\n",ret);

    /* func1 puts A6 into D0 and returns it */
  ret = MyFunc1();
  printf("ret(1)=%lx\n",ret);

    /* func2 multiplies val by 2 and puts answer into D0 for return */
  val = 10L;
  ret = MyFunc2(val);
  printf("ret(2)=%ld\n",ret);

    /* close my library */
  CloseLibrary(MyBase);

}
```

Listing 10:

```
/*****************************************************************
 * mylib.h
 *
 * defines for use with mylib.library
 *
 * History:
 *    88-05-06  created
 *****************************************************************/

#include <exec/types.h>
#include <exec/libraries.h>
#include <exec/execbase.h>
#include <libraries/dosextens.h>

/* mylib library structure - this is C version of that defined in
 * mylib_def.i
 */
struct MyLibrary {
  struct Library     ml_MyLib;
  struct ExecBase    *ml_SysLib;
  struct DosLibrary  *ml_DosLib;
  APTR               ml_SegList;
  UBYTE              ml_Flags;
  UBYTE              ml_pad;
};

#define MYLIBNAME  "mylib.library"
```

Amiga Modula-2 Programming

Visual Acuity: good technique, great buzzword!

by Clark Williams

Clark has been programming for 19 years in everything from COBOL and RPG to 68000 assembler; IBMs using OS360; everything DECish, and even PICK systems. At last count he spoke 123 different computer languages (not counting dialects of Pascal, FORTRASH, etc.)... but his first love is flying. When not aloft, he writes Modula-2 programs.

This will be a collection of topics dealing with programming the Amiga using Modula-2. We will discuss several Amiga-specific topics, as well as some general aspects of programming: coding standards, design philosophy, Visual Acuity, etc.

Admittedly these discussions will be of a religious nature. Many programmers feel very strongly about coding standards.

Most views on coding standards are strongly bonded in the formative years for programmers; attending classes where their teachers have strong and unwavering opinions that are begging to be passed on to the students. These views are often passed on without the instructor feeling any particular or compelling need to point out that these views were obtained by this teacher from his/her favourite mentor while attending some ivy hall filled with fertilizer.

I will say now that the views you read here are my own and are based on my experiences in industry and academia. I offer them to you not to convince you that I'm right - or that you are wrong - but to enable you to follow my code examples and/or procedure fragments with greater ease.

You should try the examples that are the essence of these articles. Even if you know Modula-2, you should do the examples and programatically play with them to help solidify your knowledge of the Amiga.

Coding standards, when they exist at all, are one of the most strongly held views of programmers. Every programmer seems to have his own set of standards. Each punctuation mark of these personally held standards is defended with Herculean effort. Often this defense is expressed in a language filled with graphic metaphor whose colour and imagery would put even the Amiga to shame.

No doubt my standards will seem idiotic or wise depending on whether you disagree or agree with my personal standards. However, standards are important - even critical - to a properly constructed presentation of programming technique.

My standards are based on a concept called "visual acuity." If one of the purposes of structured programming is to produce programs that are readable by both machines and humans, then we must give some consideration to what "readable form" means in the presentation of programs to programmers. The presentation of the language to the programmer should allow for quick recognition of the elements of the language. A fast glance should be all that is necessary to locate a clause or a statement, etc.

Visual Acuity is a measure of the speed by which a person unfamiliar with the code but familiar with a language syntax can find an element of that language and understand the context of that element.

Many people have suggested to me that Visual Acuity is too subjective to measure reasonably. I disagree! Based on the empirical evidence of some informal experiments, I think that any language has a maximum and measurable Visual Acuity. I think that the presentation of the form which yields maximum Visual Acuity of a language can be measured and demonstrated. I also think that some of the discussions regarding some languages made in the past few years have been without consideration of "readability," i.e. Visual Acuity.

I also believe that languages can and should be compared (within the context of the task to be implemented). Visual Acuity should be an important part of that comparison.

There are many items to consider when choosing visually acute presentations, some of these are:

- Printer vs monitor presentations.

    1) Ribbon condition, Inking, etc.

- Language syntax and construction.

1) IF-THEN-ELSE statements.

2) White space.

3) Comments, use and content.

• How easy is it to use the elements of a language designed to increase the visual acuity of that language.

1) Identifier-names (e.g. Underscore usage, etc.).

What are the goals of increasing Visual Acuity?

• To reduce debugging time.

• To reduce learning time for new users of the language.

• To standardize presentations, thereby increasing communications with and/or about the language, including its structures, such as programs, modules, routines, elements, packets, etc.

If you have read any books on structured programming, you may have noticed that the words "judicious use of white space (empty lines) and comments are useful in making your code readable." That is usually the last comment you ever read from the author assisting you in making your programs more readable.

Let's start with a discussion about identifiers. Identifiers should be easy to read. Some languages use the underscore character to break up an identifier. So that identifiers like:

buffer required for task switching

would look like this:

buffer_required_for_task_switching.

If you attempt to type in this identifier you may discover that the underscore character is located in a very inconvenient place on the keyboard. You might also agree that with poor ribbon conditions the underscore may disappear and the identifier may be confused with similar identifiers.

There are several conditions that presentation formats attempting high Visual Acuity must consider. However, this is not an article on Visual Acuity so we won't go into those conditions. Rather, I have chosen a format that solves several of the problems presented by those conditions that are encountered in a typical programimg environment.

Some of these standards are "rules." Some of these standards are "guidelines." Rules are intended to be followed as rigorously as possible. They can be broken but the intent is not to do so. Guidelines are just that. They will help you when you get stuck and don't have the time to be creative. Each standard is described below as either a rule or a guideline. Not all of the standards are described in this article but as they are encountered in examples they will be classified.

Rule #0: Identifiers should have the first letter of each word/abbreviation/mnemonic capitalized and only the first letter. As an example, our identifier from above would appear as:

BufferRequiredForTaskSwitching

This means that identifiers that include words like USA would be written as:

UsaKeyboardSwitching

It is not as easy as reading clear English text, but it takes very little effort to get used to - try it!

Guideline #0: Semi-colons and their use is another hotly discussed item. I will use semi-colons as I think Dr. Wirth intended them to be used in Modula-2. The semi-colon will be used as a statement separator, *not* a statement terminator. Therefore, you will not see a semi-colon before an END statement or before an UNTIL, etc.

Guideline #1: Comments are an element of most languages that seem to be almost an afterthought in the design of the language.

Some languages are worse at commenting than others; however, Pascal and Modula-2 happen to have comments that can be easily and usefully employed in code.

As a personal method of comment and semi-colon use I will tend to group comments using semi-colons. If a comment belongs to a particular statement you will be able to tell which statement it belongs to by the use of the semi-colon: it will be placed at the end of the comment on trailing comments and at the end of the statement on leading comments.

Thus the comment in the following code fragment:

```
Fragmentation := TRUE;

(* Inform the calling routine of the error *)

Error := TRUE;
```

is in reference to the "Error := TRUE" statement since it is between the previous statement and the semi-colon. However, if we had written the code fragment as:

```
Fragmentation := TRUE

(* Inform the calling routine of the error *);

Error := TRUE;
```

the ";" associates the comment to the first statement in the code fragment: "Fragmentation := TRUE."

Again, this is a personal use of semi-colons and comments. It is something that makes code easier for me (and others) to read. If you don't wish to use this as a convention, that is your decision to make; but you will need to know it for following my code more closely.

Guideline #2: To identify a segment of code, use the comment at the beginning of the code segment and use spaces to increase the segment's visual acuity.

As an example:

```
FOR Index := 1 TO EndOfCurrentList DO
  (* Look for all stops. *)
  ...
END (* Look for all stops. *);

(*
  The following REPEAT-UNTIL loop will process the
  stops entered in the command line.  All stops
  are processed unless an error is found during
  the processing.  See functional specification
  page 3-144.
*)

REPEAT (* looking for the stops *)
  ...
UNTIL (* No stops left or an error is detected
      which can be determined by *)
      ( Error ) OR ( AllStop );
```

Guideline #3:

Probably the biggest debate in the world of coding standards concerns the IF-THEN-ELSE statement. My primary goal is to make the statement and its clauses easy to find, and to show clearly which clauses are associated with which statements.

There are two basic ways to do the IF-THEN-ELSE statement. The first is:

```
IF <boolean> THEN
  <then-clause statements>
ELSE
  <else-clause statements>
```

The second method is:

```
IF <boolean>
  THEN (* comment *)
    <then-clause statements>
  ELSE (* comment *)
    <else-clause statements>
```

I prefer the second method for the following reasons:

1) This method is, by empirical evidence, more visually acute.

2) The statement is an IF statement with two associated clauses. The statements belonging to the THEN clause should obviously be associated with that clause. When the THEN is used as the syntactic terminator of the boolean expression you sometimes don't know on quick examination whether the IF statement is all on one line or not.

3) Ownership of the statements should be obvious. The IF owns the THEN and the ELSE clauses and the THEN and ELSE clauses own certain statements. The indentation shown in the second method leaves no question as to which statements belong to which clauses and which clauses belong to which statements. This makes it easier to identify each segment of code and thereby increases Visual Acuity.

The TDI Modula Environment

You now have an idea of how I organize my code. Let us look at some code "à l'Amiga." Specifically, let's look at getting started using TDI's implementation of Modula-2. We will start by looking at the environment available to you when you first start up a program under Modula-2.

We will be using TDI's V3.00b implementation of Modula-2. TDI's address is 10410 Markison Road, Dallas, TX 75238, (214) 340-4942.

This will not be a forum for praising, nor for bad-mouthing TDI. I know that several of you have gotten treatment from TDI that was less than business-like. Specifically, you get no Amiga help at all. I understand your frustration, but perhaps we can deal with that problem through these writings.

A one-time note on Modula-2. There are several kinds of 'modules' in Modula-2. There are program modules, library modules and client modules. A program module is easily identifiable. It always begins with the word MODULE. A client module calls a library module. A library module is called by a client module.

Obviously, this client-library module stuff depends on what point in the execution of the program we are currently examining. You could have a program module 'A', which is a client to library module 'B', which is a client to library module 'C', etc.

All library modules have two parts:

1) The DEFINITION MODULE.

2) The IMPLEMENTATION MODULE.

Rule #1: You should always define the interface to a module/ program/ packet/ library/ function/ whatever first! The implementation of the module will be much easier when the definition is fixed first!

This does not mean that you will never change the definition of the interface you design. You may well throw the interface away and start all over again. However, you should first plan

what you want to have as inputs and outputs and then review, review, review! Write down questions and write down the answers to those questions when you find them. These kinds of things will guide you toward producing better interfaces, better modules and better code.

The Amiga Developer's Manual (Bantam Books) describes an environment for languages that plan on using AmigaDOS. Specifically it describes how both the C and assembler environments have the Exec and DOS libraries already opened for the user of the language. Neither of these libraries need be opened by the user. The expectation is clearly that all languages should open up these two libraries.

We'll start with the following program module to see whether the appropriate libraries are in fact open using TDI's Modula-2.

Purpose: To determine what libraries were opened on entry to TDI's Modula-2 environment.

The plan of attack is to check the base addresses of the libraries we are interested in. The two library base addresses we want are ExecBase and DOSBase. The definition for ExecBase is found in a module that is very important to TDI's Modula-2 called AMIGAX. We'll hear more later about that module. The second address is found in a module called DOSLibrary. The program module should check the values in these addresses as its very first control statement. If they contain zeroes then the libraries are not open. If they contain non zeroes then the libraries are already opened for you. Since writing to the screen could IMPORT a module that automagically opens the DOSLibrary (but doesn't close it for you later), we will not use screen I/O. Instead we will write a file on the disk called "AnswerFile.dat" that will contain two lines telling us the state of the library at the start of the program.

These routines will, in essence, keep house for the Amiga libraries.

This would not be necessary if we knew the state of the libraries from the documentation. But TDI does not give us any specific information on the state of Modula-2 on the Amiga when we first start out.

The program module to accomplish our goal is shown in Listing 1 and is called *CheckEnvironment*:.

When the program module is run, a file called "AnswerFile.dat" will be found on the disk in DF0:. The file can be displayed on the screen by using the "TYPE" command from a CLI. The file's contents should look like:

Exec was open.
DOS was NOT open.

Obviously, DOS is not opened automatically in the TDI Modula-2 environment. This is contrary to the standard Amiga library environment; however, it offers us an opportunity to write a library module that will define our environment as the standard indicates it should be defined.

Why can't we just always open DOSBase? Well, we could. But not by the arbitrary statement:

```
DOSBase := OpenLibrary (DOSName,0)
```

The reason is that there is a module that sometimes opens the DOS library all on its own. But this module does not bother to close the library since it can't really tell when the program is done using the module. The module is MODULE Streams.

So, to determine when to open DOS - and when not to - we would have to always include the statement:

```
IF DOSBase = SYSTEM.NULL THEN
  DOSBase :=
    Libraries.OpenLibrary(DOSName,0)
END;
```

It turns out that there are a couple of modules where this kind of haphazard programming is done (see especially the MODULE Terminal). So, what we will do instead is to build a module that will open the libraries that we designate beyond DOS and Exec and will always open DOS if, and only if, it is not already open.

That means that if we ever use the Terminal module that we will have to close DOS library before the call to Terminal, or later we close the DOS library twice. Remember that AmigaDOS keeps track of which library is open and how many times that library has been opened.

These routines will, in essence, keep house for the Amiga libraries. We will call the module that describes these routines *AmigaHousekeeping*.

Library Maintenance Routines

We need some routines to handle the opening and closing of libraries when using TDI's Modula-2 and AmigaDOS. What shall we do? First we will follow some of the rules and guidelines that were defined. One of the first was to plan what we were going to do. Let us get started then with the planning.

What kind of functionality should we have in our routines? We would like to be able to call a routine that would open the Exec and DOS libraries if they weren't already opened. We would also like a routine that would open whatever libraries we specify. Each of these routines would of course have their logical opposites: routines that will close the libraries.

Since we will be manipulating libraries, let's create a type that will be representative of the libraries we will be handling. The-

se will be the libraries that we have available to be put on the shelf of our 'workbench'. While we are at it we will need a set to tell us what libraries are already open.

```
TYPE

LibraryShelf = ( Exec, DiskFont, Dos, Graphics,
        Intuit, Layers, Translate );

OnTheShelf = SET OF LibraryShelf;
```

We don't include the mathlibrary in these because, in TDI's implementation of Modula-2, the mathlibrary is not something that you have access to as a library. It is already available to you. You don't open MathLibraryBase; the variable doesn't exist. Instead you simply use the functions as you need them.

If we had a full shelf of libraries, we would have all the libraries open. If we are doing very simple and non-Amiga-specific things we probably just want DOS and Exec open. So...

```
CONST

FullShelf    = OnTheShelf { Exec..Translate };

NormalShelf = OnTheShelf { Dos };
```

Note that we don't include Exec in the normal shelf. Recall that we found out earlier that the Exec library is always opened for us before we start. So we only check for DOS in the normal shelf.

Now we are ready to discuss a routine. This routine will remember for us which libraries were opened when we began. We want the routine to return to us a variable that will allow us to find out which libraries were "on our shelf" at the time we called the routine. This routine then will be a "procedure" function.

(Momentary aside... I think this is one of the gross errors that Dr. Wirth made with Modula-2. There is a distinct difference between PROCEDURES, which do not return values through their identifiers, and FUNCTIONS which do. Procedures may not be used in expressions. Functions may be used in expressions, etc. In Modula-2 there are Procedures and Procedure Functions. Procedure Functions are FUNCTIONS. I wish that Dr. Wirth had indeed left the key word identifiers alone between Pascal and Modula-2. It makes the code much more visually acute. "Alas, that ain't what happened.")

We can then describe the inputs and outputs of our function. We do not need any input from the calling program. We do however need to be able to check the "base" addresses of the libraries. This will require that the function be able to get to the values (i.e. IMPORT these base addresses) of the base addresses of the libraries. We will return the set showing the open libraries. We will return the null set if none of the libraries were open (an error since Exec had better be open). Otherwise we will return a set indicating which libraries were open. If the set includes a library, then it was open; if not, it was closed. Look at the function called *RememberOpenLibraries* in Listing 2 to see how this was implemented.

The second procedure will open the libraries that we've requested. We need to be able to specify which libraries should be opened and we will need to know if the libraries were opened successfully. We can specify the libraries to open through a variable passed as an actual parameter. If all the libraries are opened correctly we get a TRUE back as the function value.

The problem (as always) is handling an error. If the libraries aren't opened correctly, we want to know which libraries failed to open. We can do that by letting the actual parameter serve a dual purpose. On input, it specifies what libraries are to be opened; on output, it will indicate the libaries that failed to open, if an error occurred, or return the null set if there were no errors. This design approach requires that the actual parameter be a call-by-reference parameter.

The procedure-function to accomplish this is called *OpenLibraries* and is listed in the module in Listing 2. Note that we have used the {SET} - {SET} and {SET} + {SET} constructions in the procedure-functions so far. We could also use the INCL ({SET},element) constructs as well. However I like the predicate calculus form of these equations better than the function examples. I think they are more clear and more Visually Acute. You may interchange these constructs according to your preference as long as you keep some modicum of consistency in their use.

The *CloseLibraries* procedure-function on inspection of the requirements is the same as the *OpenLibraries* procedure-function. It requires the same inputs and outputs so the procedure-function looks nearly the same.

The next procedure-function is the routine that needs to open up the Amiga libraries for us as we would normally use the libraries. That is, it should always make sure that DOS is opened, and any other libraries that we specify in an actual parameter supplied to the procedure-function. The close routine will look almost the same; however, the value returned in the actual parameter will be slightly different than that which we have described in the routines above. The parameter will not return to us the libraries that failed to open, but will return the libraries that were opened correctly whether there was an error or not.

It is not usually good practice to use two different calling conventions for similar routines; however, many times that is what must be done. The problem arises when looking at the nature of routines and modules. Modules are a collection of routines that are similar in what they work on or do, for example a math module or graphics module. It is my preferred design guideline to use a logical routine interface, rather than try to be consistent in all routines within a module. In my example, the interface to *OpenLibraries* and *AmigaOpenHouse* are

quite different, but they still are in the same module because they are logically grouped.

Our routine then is called *AmigaOpenHouse*. The idea is to open everything that is specified. The code that accomplishes this takes whatever the *RememberOpenLibraries* returns (libraries already opened), deletes these elements from the open library request, deletes exec (it should be already opened) and then opens the rest. Our calling value is left undisturbed unless there was an error in opening a library. In this case the value returned is the libraries that were opened and a FALSE value for the function.

The module initialization requires no code.

That is the module. The program module in Listing 2 shows how to use (call) the routines.

Listing 1: CheckEnvironment

```
MODULE CheckEnvironment;

FROM SYSTEM     IMPORT ADR, NULL;
FROM DOSLibrary IMPORT DOSBase, DOSName;
FROM DOSFiles   IMPORT Open,        Close,  FileLock,
                       FileHandle,  Lock,   Unlock,
                       AccessWrite, ModeNewFile,
                       Write;
FROM Libraries  IMPORT OpenLibrary, CloseLibrary;
FROM AMIGAX     IMPORT ExecBase;
FROM Strings    IMPORT Assign, Concat, Length;

CONST
 (* octal for the ANSI ASCII line feed.       *)
 LineFeed       = 12C;
 (* octal for the ANSI ASCII carriage return. *)
 CarriageReturn = 15C;
 ExecHeader     = "Exec was";
 DosHeader      = "DOS was";
 NotHeader      = " NOT";
 OpenHeader     = " open.";
 AnswerFileName = "df0:AnswerFile.dat";

TYPE
 CharArrayPtr = POINTER TO CharArray;
 CharArray    = ARRAY [ 0..79 ] OF CHAR;

VAR
 ExecAlreadyOpened : BOOLEAN;
 DosAlreadyOpened  : BOOLEAN;
 StringArray       : CharArray;
 CharBuffer        : CharArrayPtr;
 AnswerFileHandle  : FileHandle;
 Error             : LONGINT;

BEGIN (* Check Enviornment *)
 ExecAlreadyOpened := ExecBase <> NULL;
 DosAlreadyOpened  := DOSBase  <> NULL;

IF NOT DosAlreadyOpened
  THEN DOSBase := OpenLibrary ( DOSName, 0 )
 END;

 AnswerFileHandle := Open ( AnswerFileName, ModeNewFile );

 IF AnswerFileHandle <> FileHandle ( NULL )
  THEN (* file was opened successfully *)
   (*
     Set up a string to say "Exec was open."
     or "Exec was NOT open."
   *)
   Assign ( StringArray, ExecHeader );
   IF NOT ExecAlreadyOpened
    THEN Concat ( StringArray, NotHeader, StringArray )
   END;

   Concat ( StringArray, OpenHeader, StringArray );

   (*
     Stick on the carriage return for esthetics.
     Keep in mind that Length is a count of characters
     but that the array starts with index zero.
   *)

   StringArray [ Length ( StringArray ) ] := CarriageReturn;
   StringArray [ Length  (StringArray ) ] := LineFeed;

   (* Set up for Amiga write. *)

   CharBuffer := ADR ( StringArray [ 0 ] );
   Error := Write ( AnswerFileHandle,
                    CharBuffer,
                    LONG ( Length ( StringArray ) ) );
   (*
     Set up a string to say "DOS was open."
     or "DOS was NOT open.".
   *)

   Assign ( StringArray, DosHeader );

   IF NOT DosAlreadyOpened
    THEN Concat ( StringArray, NotHeader, StringArray )
   END (* of IF *);

   Concat ( StringArray, OpenHeader, StringArray );

   (*
     Stick on the carriage return for esthetics.
     Keep in mind that Length is a count of characters
     but that the array starts with index zero.
   *)

   StringArray [ Length ( StringArray ) ] := CarriageReturn;
   StringArray [ Length ( StringArray ) ] := LineFeed;

   (* Set up for Amiga write. *)

   CharBuffer := ADR ( StringArray [ 0 ] );

   Error := Write ( AnswerFileHandle,
                    CharBuffer,
                    LONG ( Length ( StringArray ) ) );
   Close ( AnswerFileHandle )
 END (* of IF *)
END CheckEnvironment.
```

Listing 2: AmigaHouseKeeping

```
DEFINITION MODULE AmigaHousekeeping;

FROM SYSTEM           IMPORT NULL;
FROM Libraries        IMPORT CloseLibrary,   OpenLibrary;
FROM AMIGAX           IMPORT ExecBase;
FROM DiskFontLibrary  IMPORT DiskFontBase,   DiskFontName;
FROM DOSLibrary       IMPORT DOSBase,        DOSName;
FROM GraphicsLibrary  IMPORT GraphicsBase,   GraphicsName;
FROM Intuition        IMPORT IntuitionBase,  IntuitionName;
FROM LayersLibrary    IMPORT LayersBase,     LayersName;
```

```
FROM TranslatorLibrary IMPORT TranslatorBase, TranslatorName;

TYPE

 LibraryShelf = ( Exec, DiskFont, Dos, Graphics, Intuit, Layers, Translate);
 OnTheShelf    = SET OF LibraryShelf;

CONST

 FullShelf   = OnTheShelf { Exec, DiskFont, Dos,    Graphics,
                            Intuit,   Layers, Translate };
 NormalShelf = OnTheShelf { Dos };

PROCEDURE RememberOpenLibraries () : OnTheShelf;
PROCEDURE OpenLibraries   ( VAR LibraryCart : OnTheShelf ) : BOOLEAN;
PROCEDURE CloseLibraries  ( VAR LibraryCart : OnTheShelf ) : BOOLEAN;
PROCEDURE AmigaOpenHouse  ( VAR LibraryCart : OnTheShelf ) : BOOLEAN;
PROCEDURE AmigaCloseHouse ( VAR LibraryCart : OnTheShelf ) : BOOLEAN;

END AmigaHousekeeping.

IMPLEMENTATION MODULE AmigaHousekeeping;

FROM SYSTEM                IMPORT NULL;
FROM Libraries             IMPORT CloseLibrary,   OpenLibrary;
FROM AMIGAX                IMPORT ExecBase;
FROM DiskFontLibrary       IMPORT DiskFontBase,   DiskFontName;
FROM DOSLibrary            IMPORT DOSBase,        DOSName;
FROM GraphicsLibrary       IMPORT GraphicsBase,   GraphicsName;
FROM Intuition             IMPORT IntuitionBase,  IntuitionName;
FROM LayersLibrary         IMPORT LayersBase,     LayersName;
FROM TranslatorLibrary     IMPORT TranslatorBase, TranslatorName;

(*
   Global TYPES that can be found in the DEF module are repeated here
   for clarity.
*)

(*
TYPE
 LibraryShelf = ( Exec,     DiskFont, Dos,
                  Graphics, Intuit,   Layers, Translate );

 OnTheShelf = SET OF LibraryShelf;

CONST
 FullShelf = OnTheShelf { Exec, DiskFont, Dos,    Graphics,
                          Intuit,   Layers, Translate };

 NormalShelf = OnTheShelf { Dos };
*)

(* End of the Global TYPES found in DEF module. *)

PROCEDURE RememberOpenLibraries () : OnTheShelf;

VAR
 LibraryBook : LibraryShelf;
 LibraryCart : OnTheShelf;

BEGIN (* Remember Open Libraries *)

 (* Initialization for the sum of elements. *)

 LibraryCart := OnTheShelf {};

(*
  Check each of the possible libraries on the shelf and see what is
  there.
*)

 FOR LibraryBook := Exec TO Translate DO
  CASE LibraryBook OF
```

```
   Exec      : IF ExecBase <> NULL
                THEN (* add exec lib to the cart we will return *)
                 LibraryCart := LibraryCart + OnTheShelf { Exec }
               END |

   DiskFont  : IF DiskFontBase <> NULL
                THEN
                 (* add disk font lib to the cart we will return *)
                 LibraryCart := LibraryCart + OnTheShelf { DiskFont }
               END |

   Dos       : IF DOSBase <> NULL
                THEN (* add dos lib to the cart we will return *)
                 LibraryCart := LibraryCart + OnTheShelf { Dos }
               END |

   Graphics  : IF GraphicsBase <> NULL
                THEN (* add graphics lib to the cart we will return *)
                 LibraryCart := LibraryCart + OnTheShelf { Graphics }
               END |

   Intuit    : IF IntuitionBase <> NULL
                THEN
                 (* add intuition lib to the cart we will return *)
                 LibraryCart := LibraryCart + OnTheShelf { Intuit }
               END |

   Layers    : IF LayersBase <> NULL
                THEN (* add layers lib to the cart we will return *)
                 LibraryCart := LibraryCart + OnTheShelf { Layers }
                END |

   Translate : IF TranslatorBase <> NULL
                THEN
                 (* add translator lib to the cart we will return *)
                 LibraryCart := LibraryCart + OnTheShelf { Translate }
                END |

   ELSE (* DO NOTHING *)
  END (* of CASE *)
 END (* of FOR *);

 RETURN LibraryCart

END RememberOpenLibraries;

PROCEDURE OpenLibraries ( VAR LibraryCart : OnTheShelf ) : BOOLEAN;

VAR
 LibraryBook : LibraryShelf;

BEGIN (* Open Libraries *)
 IF NOT ( LibraryCart = OnTheShelf {} )
  THEN
   FOR LibraryBook := Exec TO Translate DO
    CASE LibraryBook OF
     Exec      : IF Exec IN LibraryCart
                  THEN
                   ExecBase    := OpenLibrary ( "exec.library", 0 );
                   LibraryCart := LibraryCart - OnTheShelf { Exec }
                 END |

     DiskFont  : IF DiskFont IN LibraryCart
                  THEN
                   DiskFontBase := OpenLibrary ( DiskFontName, 0 );
                   LibraryCart  :=   LibraryCart
                                   - OnTheShelf { DiskFont }
                 END |

     Dos       : IF Dos IN LibraryCart
                  THEN
                   DOSBase     := OpenLibrary ( DOSName, 0 );
                   LibraryCart := LibraryCart - OnTheShelf { Dos }
```

```
                END |

    Graphics  : IF Graphics IN LibraryCart
                 THEN
                  GraphicsBase := OpenLibrary ( GraphicsName, 0 );
                  LibraryCart  :=   LibraryCart
                                  - OnTheShelf { Graphics }
                END |

    Intuit    : IF Intuit IN LibraryCart
                 THEN
                  IntuitionBase := OpenLibrary ( IntuitionName, 0 );
                  LibraryCart   :=   LibraryCart
                                   - OnTheShelf { Intuit }
                END |

    Layers    : IF Layers IN LibraryCart
                 THEN
                  LayersBase  := OpenLibrary ( LayersName, 0 );
                  LibraryCart := LibraryCart - OnTheShelf { Layers }
                END |

    Translate : IF Translate IN LibraryCart
                 THEN
                  TranslatorBase := OpenLibrary (TranslatorName, 0);
                  LibraryCart    :=   LibraryCart
                                    - OnTheShelf { Translate }
                END |

    ELSE (* DO NOTHING *)
   END (* of CASE *)
  END (* of FOR *);
  RETURN LibraryCart = OnTheShelf {}
 END (* of IF *)

END OpenLibraries;

PROCEDURE CloseLibraries ( VAR LibraryCart : OnTheShelf ) : BOOLEAN;

VAR
 LibraryBook : LibraryShelf;

BEGIN (* Close Libraries *)
 IF NOT ( LibraryCart = OnTheShelf {} )
  THEN
   FOR LibraryBook := Exec TO Translate DO
    CASE LibraryBook OF

    Exec      : IF Exec IN LibraryCart
                 THEN
                  IF ExecBase <> NULL
                   THEN CloseLibrary ( ExecBase )
                  END;
                  LibraryCart := LibraryCart - OnTheShelf { Exec }
                END |

    DiskFont  : IF DiskFont IN LibraryCart
                 THEN
                  IF DiskFontBase <> NULL
                   THEN CloseLibrary ( DiskFontBase )
                  END;
                  LibraryCart :=   LibraryCart
                                 - OnTheShelf { DiskFont }
                END |

    Dos       : IF Dos IN LibraryCart
                 THEN
                  IF DOSBase <> NULL
                   THEN CloseLibrary ( DOSBase )
                  END;
                  LibraryCart := LibraryCart - OnTheShelf { Dos }
                END |

    Graphics  : IF Graphics IN LibraryCart
                 THEN
                  IF GraphicsBase <> NULL
                   THEN CloseLibrary ( GraphicsBase )
                  END;
                  LibraryCart :=   LibraryCart
                                 - OnTheShelf { Graphics }
                END |

    Intuit    : IF Intuit IN LibraryCart
                 THEN
                  IF IntuitionBase <> NULL
                   THEN CloseLibrary ( IntuitionBase )
                  END;
                  LibraryCart := LibraryCart - OnTheShelf { Intuit }
                END |

    Layers    : IF Layers IN LibraryCart
                 THEN
                  IF LayersBase <> NULL
                   THEN CloseLibrary ( LayersBase )
                  END;
                  LibraryCart := LibraryCart - OnTheShelf { Layers }
                END (* CASE is continued on next page *)|

    Translate : IF Translate IN LibraryCart
                 THEN
                  IF TranslatorBase <> NULL
                   THEN CloseLibrary ( TranslatorBase )
                  END;
                  LibraryCart :=   LibraryCart
                                 - OnTheShelf { Translate }
                END |

    ELSE (* DO NOTHING *)
   END (* of CASE *)
  END (* of FOR *);
  RETURN LibraryCart = OnTheShelf {}
 END (* of IF *)
END CloseLibraries;

PROCEDURE AmigaOpenHouse ( VAR LibraryCart : OnTheShelf ) : BOOLEAN;

(*
  In normal Amiga house keeping the exec library is already opened. *)

VAR
 LibrariesOpened : OnTheShelf;

BEGIN (* Amiga Open House *)
(* Now we open the libraries we need. *)
 LibraryCart :=   FullShelf
                - LibraryCart (* ones we want. *)
                - RememberOpenLibraries () (* ones already open *)
                - OnTheShelf { Exec } (* exec always open. *)     ;

 (* Calculate the libraries that we opened. *)

 LibrariesOpened := FullShelf - LibraryCart;

 RETURN OpenLibraries ( LibrariesOpened )

END AmigaOpenHouse;

PROCEDURE AmigaCloseHouse ( VAR LibraryCart : OnTheShelf ) : BOOLEAN;

(*
  In normal Amiga house keeping the exec lib is closed automagically. *)

BEGIN (* Amiga Close House *)
 RETURN CloseLibraries ( LibraryCart )
END AmigaCloseHouse;

BEGIN (* Amiga Housekeeping *)
END AmigaHousekeeping.
```

Programming in M2AMIGA

Translating C source

by Anthony Bryant

Anthony Bryant is a graduate of the University of Manitoba. He is a computer engineer working with remote control systems and robotics, and programs the Amiga in his spare time.

Both professional programmers and weekend hackers have one thing in common, guaranteed to get the juices flowing. No, not that! I am talking about a new language implementation. Studying a new language can offer many new insights that can be useful in furthering ones programming art.

This article details some of my experiences in adapting and translating C source (all in the public domain) to the M2AMIGA Development System.

Since Modula-2 (herein refered to as M2) is a newer language than C, most of the important support libraries (which are the life-blood of Amiga) will be adaptations of the original C source. Fortunately, M2AMIGA adheres to the C nomenclature in the libraries, simplifying translation. Separate compilation of modules during development along with a fast compiler and linker allowed for speedy translation.

One of the first library modules I worked on was Rob Peck's well-known "AudioTools," a comprehensive collection of simple-to-use audio routines that enable the user to queue up sounds (PlayFreq, PlayNote and PlaySamp) and have the audio device play them, sequentially, while your program continues doing what it was doing. Rob sent me his latest version in C, Release 3.0, and I set to work. I soon came to appreciate the well commented source (which lives on in the M2 version - see AudioTools.def and AudioTools.imp). It's important to know not just what the code does, but also what the designer had in mind when he wrote it. Rob's source is also an excellent tutorial on using IORequests with devices.

Required adjustments

Since M2AMIGA's compiler produces code in one pass, all referenced objects must already be known. If a procedure is to be called whose declaration is further down the program, the procedure must be FORWARD declared. (FORWARD is a reserved word). I opted, instead, to re-order all of the procedures so that this wasn't necessary, and in the process of jotting down all of the procedures called by other procedures, I gained an overview of the structure of the program which proved helpful in adding and changing some routines.

The next step was to check the interface with all of the Amiga Library routines; in this case, from "Dos" and "Exec." M2 doesn't permit procedures that are both function and proper procedures. For example, in C you could have:

```
error=WaitIO(iob);/*a function procedure*/
```

or

```
WaitIO(iob);        /* a proper procedure */
error = iob->Request.io_Error;
```

Both examples return the error which has been placed in io_Error field after the WaitIO has completed. But in M2 the latter case is the only choice.

Also, you might find in C:

```
dummy = WaitPort(uport);
```

while in M2, WaitPort is a proper procedure, so there is no need of a dummy variable.

Flags, operators and constructs

The representation of flags in M2 is slick, compared to C. Basic types, SHORTSET, BITSET, and LONGSET handle the packing of flag bits into 8 bits, 16 bits and 32 bits respectively. For example, the IORequest flags are given the name IOFlagSet (defined as a SHORTSET; i.e., bits 0 to 7) with:

```
quick  = IOFlagSet{0};  i.e. bit 0 is set
noWait = IOFlagSet{6};  i.e. bit 6 is set
```

The + and - operators then are used to set or clear flags. M2 really has none of C's bitwise operators and none are needed. You can easily check to see if bits are set or clear. In addition, you can define your own SETs as subsets of the basic types.

M2 has a nice selection of looping statements to chose from. Decisions, decisions! In C, the 'for' looping construct is really a variation on the basic 'while' construct. In adapting from C,

I have found that, in most cases, if the 'for' index variable post-increments or decrements, I would chose M2's FOR statement, but if the 'for' index variable pre-increments or decrements, I would use M2's WHILE statement. C's 'do .. while' becomes M2's REPEAT .. UNTIL, but don't forget to reverse the logic of the 'while' expression! The one use of a 'goto' in AudioTools was easily handled by M2's LOOP and EXIT combination.

String operations

M2's definition of strings is slightly different from C's. If you want to pass a string parameter to an Amiga Library routine, that string must have a terminating null character. (In M2 either 0C or "".) But a variable defined in M2, say as:

```
VAR
  string: ARRAY [0..11] OF CHAR;
```

could be filled completely with 12 chars and *no* null char. (If less than 12 then, of course, there will be a terminating null.) A special feature of the M2AMIGA compiler assures that all constant strings, stored in the constant space in a program, are properly terminated by a null. Thus:

```
OpenDevice(ADR("audio.device"), ....... )
```

or

```
CONST audioName="audio.device"
OpenDevice(ADR(audioName), ...... )
```

M2AMIGA provides the requisite procedures for string manipulation - Compare(), Insert(), Delete(), Occurs(), etc - handling string assignments with ease.

Numeric operations and addresses

Numeric constants, likewise, are handled with ease. All constants, being untyped, are converted (coerced) into the type required by the library routine.

Integer arithmetic is another area where slight differences exist. Rob's routines for calculating note duration and cycles already maximize precision while taking care not to overflow a 32-bit word. In M2, an unsigned short is called a CARDINAL while an unsigned long is called a LONGCARD. Because of M2's strict type-checking, it's happiest when multiplying or dividing the same types. It won't let you multiply a signed integer, INTEGER or LONGINT, by an unsigned type. If you know (or suspect) that the product of two CARDINALs is going to be a LONGCARD (very likely!) you are best to do a type conversion. (M2AMIGA does type conversions rather than type transfers [M2 lingo] since this is more secure. If you don't, M2AMIGA's run-time system will alert you to an "Overflow" error!)

(As an aside: Once you've finished debugging, you can switch off the run-time system checking. This stops the compiler from generating chunks of code that check: subrange and array indexing; overflow and underflow; division by zero; stack overflow check; and more ....)

M2AMIGA defines ADDRESS as a pointer to a BYTE.

```
TYPE
  ADDRESS = POINTER TO BYTE;
```

ADDRESS is type-compatible with LONGINT, so that when you de-reference an ADDRESS you are looking at a BYTE, and when you INCrement or DECrement an ADDRESS, it is by the BYTE.

But in C, you can have (BYTE *) and (WORD *), both addresses, but what a difference! Without exception, you can translate (BYTE *) as ADDRESS, and ARRAYS OF BYTE are used in preserving audio waveform data in AudioTools.

The use of (WORD *) is more complex, since it is usually accompanied by the indexing [] of an address. The indexing is by the WORD. C goes along with this, hoping that the C programmer knows what he is doing! (Indexing an address, indeed!) In M2, you can pass open ARRAYS OF WORD, which is really what we are dealing with in this case, between procedures. The use of open ARRAYS greatly simplifies this task (and in sorting out pointers-to-pointers). The actual ARRAY is not duplicated, as this would obviously consume more space; instead, only its address and length are passed, ensuring the integrity of the data.

M2's strict type-checking forces you to think about what you are doing and about the results of your actions, rather than leaving it up to the compiler to figure out what you meant. It clears away all (well, most) of the cobwebs, and you are far less likely to be bitten by a hidden bug later on.

Most of the adaptation work, though, was minor editing; i.e., changing '/*' and '*/' to '(*' and '*)', changing '=' to ':=' and '==' to '=', changing '!=' to '#' (I like # better than <>), and changing '->' to '^'. M2 has special commands, INC and DEC, to replace '++', '+=', '--' and '-='. SHIFT handles '<<' and '>>', MOD is '%', NOT is '!', AND is '&&', OR is '||'. (It recalls to mind EASY C May 1986 BYTE magazine!) Done! (Well, almost...)

The feature I missed most is C's compiler directive, #define, which, besides defining constants, is very often used to replace entire procedures, both function and proper. In AudioTools, this technique is used effectively to add all of the Channel commands, which are all defined in terms of the basic ControlChannel() command. No choice here but to write out many very short procedures.

The AudioTools module (on disk) has the appearance and function of any other library module; consisting of a symbol file (sym/AudioTools.sym) that the compiler uses, and an object file (obj/AudioTools.obj) that the linker uses. One of the reasons M2AMIGA compiles and links as fast as it does is these

"pre-compiled" modules. All of the symbol files needed are brought into RAM during compilation, and all of the object files needed are likewise brought into RAM during linking.

The test program, AudioDemo.mod, is a simple example that IMPORTs all of the procedures FROM AudioTools. In order to have some waveforms to play, I first set up some wave arrays - sawtooth, triangle, and square. Although M2AMIGA has extensive math libraries in REALS, LONGREALS, transcendentals and IEEE double precision transcendentals, I chose M2AMIGA's special fast floating point type, FFP, and simulated my sines and cosines. Conversion of FFP to INTEGER (and vice-versa) is simple enough and voila - a sinewave array. (I also tested these in AmigaBasic - obtaining a plot to see that the start of the waveform equalled the end of the waveform, somewhat important in linking audio data waves to avoid clicks and other noise.) (AudioTools expands the basic 256 byte array to achieve higher frequencies and these waves are linked end-to-end - see SetWave() in AudioTools.imp.)

The use of M2AMIGA's Terminal library module greatly simplifies a short test program like this. As M2AMIGA supports both CLI and Workbench, Terminal automatically opens up a window (if called from Workbench) on which to write strings (or write decimal or hex numbers). M2AMIGA's run-time system hides some of the grunt work from the programmer by executing startup, automatically opening and closing Amiga libraries, and passing CLI or WB args. (This is a liability if you absolutely need a non-standard AStartup.)

M2AMIGA's Arts library module forms the basis of all programs and is integrated with its other library modules. (Not Amiga Library modules). When run-time errors occur during development, they do not go the way of the guru. Instead, the program is interrupted with an AutoRequester which reports the cause of the error and leads to the freeing up of all allocated resources that it knows about through its other library modules (by the use of a TermProcedure(), see below). For example, aborting after the error requester closes the Terminal window (if open), the FileSystem module closes all open files, the Heap module deallocates all previously allocated memory, etc.

The Arts module contains several interesting procedures, such as BreakPoint(), which causes a (temporary) program interrupt, and DebugProcedure() which calls (installs) a debugger (not supplied) following a program crash or interrupt. You can even add information to be displayed in the error requester or design your own debugger using information about the crash/interrupt stored in Arts very own variable, errorFrame.

Of course, you don't have to use any of it (nobody is twisting your arm) and so, since AudioTools uses mainly the Amiga Library modules, I decided to use only Arts' TermProcedure() to do my own cleanup - that is to call FinishAudio(). TermProcedure() functions at the end of the program and in case of error (and, after debugging, could really be removed - just don't forget to call FinishAudio()!). This avoids the need to reset the Amiga after each debugging session!

And finally, the code generated to produce the executable - AudioDemo - is as compact as that of Manx C, though it contains extra stuff (Terminal window, etc).

AudioTools makes a nice addition to anyone's library, whether in C or M2 and thanks to Rob's source, I learned a lot about working directly with Amiga devices.

Having a PlaySamp and PlayNote in my library, I turned my gaze to a ReadSamp and a ReadSmus, to be found in the IFF specification 8SVX and SMUS. But that's a whole other story (and a much larger library).

```
DEFINITION MODULE AudioTools;
(* ---------------------------------------------------------------------------
 * BASED ON AUDIOTOOLS version RELEASE.3.0  by  Rob Peck
 *
 * This software is in the public domain
 *
 * M2Amiga Modula-2 adaptation by Anthony Bryant
 * ---------------------------------------------------------------------------*)
FROM SYSTEM IMPORT
 ADDRESS, BYTE;
FROM Audio IMPORT
 hardChannels;
FROM Exec IMPORT
 Byte, IORequest, Message, MsgPortPtr;

CONST
 maxChan = hardChannels;
 audBuffers = 20;  (* number of iob msg packets before need to allot *)
 audClock = 3579545;
                          (* error return messages *)
 badChannelSelected = -1;  (* channel # out of range *)
 notYourChannel     = -2;  (* not owned by your task *)
 outOfMemory        = -3;  (* error return from AllocMem *)

TYPE
 ExtIOB=RECORD
  request:  IORequest;
  allocKey: INTEGER;
  data: ADDRESS;
  length: LONGCARD;
  period: CARDINAL;
  volume: CARDINAL;
  cycles: CARDINAL;
  writeMsg: Message; (* up to here, same as IOAudio *)
  identifier: LONGINT;  (* This field is added *)
 END;
 ExtIOBPtr=POINTER TO ExtIOB;


PROCEDURE InitAudio(): MsgPortPtr;
PROCEDURE CheckIfDone(): BOOLEAN;
PROCEDURE FinishAudio(uport: MsgPortPtr);

PROCEDURE GetChannel(channel: LONGINT): LONGINT;
PROCEDURE FreeChannel(channel: LONGINT): LONGINT;
PROCEDURE StopChannel(channel: LONGINT): LONGINT;
PROCEDURE StartChannel(channel: LONGINT): LONGINT;
PROCEDURE FlushChannel(channel: LONGINT): LONGINT;
PROCEDURE ResetChannel(channel: LONGINT): LONGINT;
PROCEDURE IsThatMyChan(channel: LONGINT): LONGINT;

PROCEDURE MayGetNote(uport: MsgPortPtr; flag: BOOLEAN): LONGINT;

PROCEDURE SetWave(channel: LONGINT;
            VAR waveform: ARRAY OF BYTE): LONGINT;
PROCEDURE SetSamp(channel: LONGINT;
            VAR sampleaudio: ADDRESS;
                length: LONGINT): LONGINT;
PROCEDURE SetPV(channel: LONGINT;
              period, volume: CARDINAL): LONGINT;
```

```
PROCEDURE PlayNote(channel: LONGINT;
                   note:    CARDINAL;  (* specific note number 0-95 *)
                   volume:   CARDINAL; (* volume 0-64 *)
                   duration: CARDINAL; (* 1000ths of a second *)
                   priority: Byte; (* precedence value -128 to 127 *)
                   messageport: MsgPortPtr; (* for use by MayGetNote() *)
                   id: LONGINT);
PROCEDURE PlayFreq(channel: LONGINT;
                   freq:   CARDINAL;   (* specific frequency 28Hz to 7000Hz *)
                   volume:  CARDINAL;  (* volume 0-64 *)
                   duration: CARDINAL; (* 1000ths of a second *)
                   priority: Byte; (* precedence value -128 to 127 *)
                   messageport: MsgPortPtr; (* for use by MayGetNote() *)
                   id: LONGINT);
PROCEDURE PlaySamp(channel: LONGINT;
                   period:  CARDINAL;  (* specific period value 127 to 500 *)
                   volume:  CARDINAL;  (* volume 0-64 *)
                   duration: CARDINAL; (* 1000ths of a second *)
                   priority: Byte; (* precedence value -128 to 127 *)
                   messageport: MsgPortPtr; (* for use by MayGetNote() *)
                   id: LONGINT);

END AudioTools.def



IMPLEMENTATION MODULE AudioTools;
(* ---------------------------------------------------------------------------
 * BASED ON AUDIOTOOLS version RELEASE.3.0  by  Rob Peck
 *
 * This software is in the public domain
 *
 * M2Amiga Modula-2 adaptation by Anthony Bryant
 * ---------------------------------------------------------------------------*)
FROM SYSTEM IMPORT
 ADDRESS,ADR,BYTE,LONGSET;
FROM Audio IMPORT
 free, perVol, allocate,                (* ADCMD_  commands *)
 pervol, syncCycle, noWait, writeMessage,   (* ADIOF_  flags *)
 IOAudio;
FROM Dos  IMPORT
 Delay;
FROM Exec IMPORT
 invalid,reset,read,write,update,clear,stop,start,flush, (* IOAudio cmds *)
 quick, (* IOF_ flags *)
 IORequest, Message, MsgPortPtr, Node, TaskPtr, UnitPtr, DevicePtr,
 UByte, Byte, MemReqs, MemReqSet,
 AllocMem, CloseDevice, FindTask, FreeMem, GetMsg, OpenDevice,
 PutMsg, WaitIO, WaitPort;
FROM ExecSupport IMPORT
 BeginIO, CreatePort, DeletePort;


CONST
 waveSize=512; (* byte size of allocated memory for waves ONLY *)

TYPE
 auMsg=RECORD
  message: Message;
  identifier: LONGINT;   (* matches the bottom of ExtIOB *)
 END;
 auMsgPtr=POINTER TO auMsg;

VAR
 unit: ARRAY [0..maxChan-1] OF UnitPtr;  (* global pointers to Units *)
 key:  ARRAY [0..maxChan-1] OF INTEGER;  (* global value for alloc keys *)
 usertask: ARRAY [0..maxChan-1] OF TaskPtr; (* user owns which channels *)
 (* in preparation for making this a shared library (loadable from disk) *)

 openIOB: IOAudio;        (* IOB to open and close the device *)
 device: DevicePtr;       (* global pointer to audio device   *)
 controlPort: MsgPortPtr; (* Port for ControlChannel functions *)

 audbuffer: ARRAY [0..audBuffers-1] OF ExtIOB;  (* global, static buffers    *)
 inuse:     ARRAY [0..audBuffers-1] OF BOOLEAN;(* keep track of statics used *)
 chipaudio: ARRAY [0..maxChan-1] OF ADDRESS;   (* pntrs to waves in CHIP RAM *)
 datalength: ARRAY [0..maxChan-1] OF LONGINT;  (* length of data in CHIP RAM *)
 replyPort: ARRAY [0..maxChan-1] OF MsgPortPtr;   (* one ReplyPort per chan  *)
 dynamix:   ARRAY [0..maxChan-1] OF LONGINT;  (* keep track of dynamics used *)
 anychan:     ARRAY [0..maxChan-1] OF UByte;  (* channel masks for mono *)

 dynamicName: ADDRESS; (* "dynamic" IOB's *)
 globalName: ADDRESS;  (* "global" ( really "static") IOB's *)

    (* Each waveform buffer contains 8 octaves of the wave.  *)

woffsets: ARRAY [0..8] OF CARDINAL; (* where waveform for that octave begins.*)
wlen: ARRAY [0..8] OF CARDINAL;   (* length of each waveform within a buffer *)
perval: ARRAY [0..12] OF CARDINAL; (* Period of these notes within an octave *)



(*------------------- internal   procedures ------------------------*)

(* FreeIOB - free a global (really static) or a dynamic, allocated IOB *)

PROCEDURE FreeIOB(iob: ExtIOBPtr; channel: LONGINT);
 VAR
   i: CARDINAL;
 BEGIN
   IF (iob^.request.message.node.name = dynamicName) THEN
     FreeMem(iob, SIZE(iob^));
     IF (dynamix[channel] # 0) THEN
       DEC(dynamix[channel]); (* subtract one if nonzero *)
     END;
   ELSIF (iob^.request.message.node.name = globalName) THEN
     i:= iob^.request.message.length;
     IF (i < audBuffers) THEN
   inuse[i]:= FALSE;   (* frees this one for reuse *)
     END;
   END;
 END FreeIOB;

(* ReEmployIOB - look at ALL of the reply ports and if any IOBs
 * hanging around with nothing to do, then free them.
 *
 * Audio may still be playing the waveform as we get a message
 * through MayGetNote.  MayGetNote marks the iob message block as free-able,
 * (when it finds that the identifier field is set to zero) but we have
 * to have a way of recirculating in this list of messages.
 *
 * In other words, if something is free-able, free it, otherwise leave it
 * on the list.  So rather than removing things from the front of the list,
 * lets just walk through the message list, remove (dequeue) what is
 * freeable and leave the rest there to look at the next time.
 *)

PROCEDURE ReEmployIOB();
 VAR
   i: LONGINT;
   mp: MsgPortPtr;
   iob: ExtIOBPtr;
   pushback: ExtIOBPtr;

   (* What happens here is that iob's are removed from the message port
    * when they come back from the audio device.   If YOU have set the
    * messageport nonzero, it means that you wanted to know when
    * this note began to play.  The WriteMsg part of the iob is then
    * linked, as a message, onto your user port.  So this routine
    * cannot free the iob until it is certain that YOU have finished
    * using it.  The iob_Priority field is READ here.  If it still
    * nonzero, the iob is pushed back onto the message port (on the
    * end of the message queue) to be read again.  We hold a pointer
    * named "pushback" that lets us keep track of when we see that
    * again.  If we see it twice, it means we have completed a full
    * circle through the queue of messages and have freed everything
    * that we can this time.  Therefore, we examine it and either
    * free it or push it back again, then exit.
    *)

 BEGIN
   FOR i:=0 TO maxChan-1 BY 1 DO
      (* remove all iob's from ALL ports, unless we have to push one back *)
      mp:= replyPort[i];

      pushback:= NIL;   (* nothing pushed back so far *)
```

```
      iob:= ExtIOBPtr(GetMsg(mp));
      WHILE (iob # NIL) DO
    (* First see if messageport in writeMsg is not NIL;          *)
    (* if so, audio device is done, but user has not acknowledged *)
    (* this message yet (by using MayGetNote). *)

   IF (iob^.writeMsg.replyPort # NIL) THEN
      PutMsg(mp, iob);
      IF ((iob # pushback) AND (pushback = NIL)) THEN
            pushback:= iob;  (* Remember FIRST one pushed back *)
      END;
   ELSE
          FreeIOB(iob,i); (* messageport is NIL, can free the iob *)
   END;
      iob:= ExtIOBPtr(GetMsg(mp));
      END;
   END;
END ReEmployIOB;

(* GetIOB - allocate an IOB , global (really static)  or dynamic for use. *)

PROCEDURE GetIOB(channel: LONGINT): ExtIOBPtr;
 VAR
   i, usereply: CARDINAL;
   iob: ExtIOBPtr;     (* in case we need to allocate one *)
 BEGIN
   ReEmployIOB();      (* find already used ones and free them *)

   IF (channel = -1) THEN usereply:= 0; ELSE usereply:= channel; END;

   (* try to allocate a global (really static) iob to use *)
   FOR i:=0 TO audBuffers-1 BY 1 DO
      IF (inuse[i] = FALSE) THEN
        (* we have our global (really static), so assign parameters *)
   inuse[i]:= TRUE;
   audbuffer[i].request.device:= device;
   audbuffer[i].request.message.replyPort:= replyPort[usereply];
   audbuffer[i].request.message.length:= i;
   audbuffer[i].request.message.node.name:= globalName;
       RETURN ADR(audbuffer[i]);
      END;
   END;

   (* if all globals (really statics) are in use,try to allocate dynamic one *)
   iob:= ExtIOBPtr(AllocMem(SIZE(iob^), MemReqSet{memClear}));
   IF (iob = NIL) THEN RETURN NIL; END;   (* out of memory *)
      (* we have our dynamic, so assign parameters *)
       iob^.request.device:= device;
       iob^.request.message.replyPort:= replyPort[usereply];
       iob^.request.message.node.name:= dynamicName;
       iob^.request.message.length:= dynamix[usereply];
       INC(dynamix[usereply]); (* add one to number allocated to a channel *)
    RETURN iob;
 END GetIOB;


(* CheckIOBDone - to see if all iob's are finished (i.e. freed up)
 * if TRUE then everything IS finished.
 *)

PROCEDURE CheckIOBDone(): BOOLEAN;
 VAR
   i, status: LONGINT;
 BEGIN
   status:= 0;   (* means there are still some iob's in play *)
                 (* when status = 4, then everything is free *)

   FOR i:=0 TO audBuffers-1 BY 1 DO
     IF (inuse[i] = TRUE) THEN
    (* Sooner or later, this will catch both
          * the statics and dynamics.  Note that
          * this will only work if NO (REPEAT: NO)
          * iob's sent off with a duration value
          * of "0", because zero means "forever"
          *)
        ReEmployIOB();
     END;
   END;
   (* Note to implementors... maintaining inuse[i] now seems
    * like a lousy idea, unless it is accompanied by a variable
    * statics_inplay that decrements to zero when all statics
    * are done.  That makes it much easier to check than going
    * through all of the inuse[]'s.  Maybe not.
    *)

   FOR i:=0 TO maxChan-1 BY 1 DO
     IF (dynamix[i] > 0) THEN
                (* If this channel still playing a   *)
                (* dynamically allocated block, wait *)
                (* for all messages to return before *)
                (* the program exits.                *)
       ReEmployIOB();  (* take another shot at freeing it all *)
     END;
   END;

   FOR i:=0 TO maxChan-1 BY 1 DO   (* Check again as we nearly exit *)
     IF (dynamix[i] = 0) THEN INC(status); END;
   END;
   IF (status = 4) THEN    (* all dynamics are free, now check the statics *)
     FOR i:=0 TO audBuffers-1 BY 1 DO
       IF (inuse[i] = TRUE) THEN RETURN FALSE; END;     (* some not free *)
     END;
     RETURN TRUE;   (* DONE! *)
   ELSE
     RETURN FALSE;  (* still some out there! *)
   END;
 END CheckIOBDone;

(* -------------- USER  support  procedures ----------------- *)

(* InitAudio returns, uport, a pointer to a message port at which your task
 *  receives a message when a particular note BEGINS to play.
 * You must save this value somewhere, and use it to call MayGetNote
 *  or FinishAudio.  MayGetNote is the name of the routine that you call
 * to check if a note has begun to play.
 *  If an error occurs (can't Opendevice  or CreatePorts) then pointer = NIL
 *)

PROCEDURE InitAudio(): MsgPortPtr;
 VAR
   error,i: LONGINT;
   firstuser: BOOLEAN;  (* THIS WILL GET MOVED when shared library is made *)
 BEGIN
   firstuser:= TRUE;

   FOR i:=0 TO audBuffers-1 BY 1 DO
     inuse[i]:= FALSE;      (* declare all message blocks are available *)
   END;

   openIOB.length:= 0;   (* Open device but don't allocate channels *)
   OpenDevice(ADR("audio.device"),0,ADR(openIOB),LONGSET{0});
   (* returns error in io_Error field; should be 0 *)
   error:= LONGINT(openIOB.request.error);  (* IOERR_OPENFAIL  -1 *)
   IF (error # 0) THEN RETURN NIL; END;
   device:= openIOB.request.device; (* Get the device address for later use *)

   FOR i:=0 TO maxChan-1 BY 1 DO
     replyPort[i]:= CreatePort(0,0); (* ports for replies from each channel *)
     IF (replyPort[i] = NIL) THEN RETURN NIL; END;

     chipaudio[i]:= 0;  (* have not yet created the waves/samples *)
     datalength[i]:= 0; (* length of wave/sample data in CHIP RAM *)
     dynamix[i]:= 0;   (* no dynamic I/O blocks allocated *)

     (* When implemented as a shared library, "firstuser" will only *)
     (* be TRUE when the library is first opened. *)

     IF (firstuser = TRUE) THEN
       key[i]:= 0;          (* init key values *)
       unit[i]:= NIL;       (* init unit values *)
       usertask[i]:= NIL;   (* no channel owned by any task *)
     END;
   END;

   controlPort:= CreatePort(0,0); (* use for control & syncronous functions *)
   IF (controlPort = NIL) THEN RETURN NIL; END;

     (* init anychan ARRAY for use by GetChannel *)
```

```
  anychan[0]:=1; anychan[1]:=2; anychan[2]:=4; anychan[3]:=8;
      (* init waveform buffer offsets ARRAY for use by PlayNote *)
  woffsets[0]:=0; woffsets[1]:=256; woffsets[2]:=384; woffsets[3]:=448;
  woffsets[4]:=480; woffsets[5]:=496; woffsets[6]:=504; woffsets[7]:=508;
  woffsets[8]:=510;
      (* init length of each waveform in a buffer ARRAY *)
  wlen[0]:=256; wlen[1]:=128; wlen[2]:=64; wlen[3]:=32; wlen[4]:=16;
  wlen[5]:=8  ; wlen[6]:=4;   wlen[7]:=2;  wlen[8]:=1;
      (* init period value to go with note within an octave *)
  perval[0]:=428; perval[1]:=404; perval[2]:=381; perval[3]:=360;
  perval[4]:=339; perval[5]:=320; perval[6]:=302; perval[7]:=285;
  perval[8]:=269; perval[9]:=254; perval[10]:=240; perval[11]:=226;
  perval[12]:=214;
      dynamicName:= ADR("dynamic");
      globalName := ADR("global");

  RETURN CreatePort(0,0); (* my user port *)
 END InitAudio;


(*---------------- USER support procedures ----------------- *)

(* GetChannel:  To request "any" channel, use channel = -1;
 * To request a specific channel, use channel = 0, 1, 2 or 3;
 * Again NOTE, this returns two globals as well as the channel number!
 *)

PROCEDURE GetChannel(channel: LONGINT): LONGINT;
 VAR
   error, channum: LONGINT;
   addrmsg: ADDRESS;
   iob: ExtIOBPtr;
   controlIOB: ExtIOB;
 BEGIN
   iob:= ADR(controlIOB);
   iob^.request.device:= device;
   iob^.request.message.replyPort:= controlPort;
   iob^.allocKey:= 0; (* zero for new key *)
   iob^.request.message.node.pri:= 20;

   IF (channel = -1) THEN
     iob^.data:= ADR(anychan[0]);
     iob^.length:= 4;
   ELSIF ((channel >=0) AND (channel < maxChan)) THEN

            (* NOTE ***** ENHANCEMENT COMING HERE ***** *)

      IF (usertask[channel] # NIL) THEN RETURN (notYourChannel); END;

   (* Enhancement might be: look at the running priority
    * of the current task as compared to the running priority
    * of the task in usertask[i].  If not same task and if
    * the current task has a higher priority, STEAL the channel!
    * Alternative (seems better) is to have a global variable
    * called audPriority to be set by a new function SetAudPriority
    * (for a given task only), and that global priority value
    * would be used for GetChannel and LockChannel requests.
    *)
     iob^.data:= ADR(anychan[channel]);
     iob^.length:= 1;

   ELSE   (* chose a bad channel number; cannot allocate it *)
     RETURN (badChannelSelected);
   END;
   iob^.request.command:= allocate;  (* ADCMD_ALLOCATE *)
   iob^.request.flags:= noWait + quick; (* ADIOF_NOWAIT | IOF_QUICK *)
   BeginIO(iob);
   WaitIO(iob);  (* returns error in io_Error field; should be 0  *)
   error:= LONGINT(iob^.request.error);  (* ADIOERR_NOALLOCATION  -10 *)
   IF (error # 0) THEN RETURN error; END;
 (* WaitIO, just above,removes the message from the port. No need of GetMsg *)

   CASE LONGINT(iob^.request.unit) OF
     1 :   channum:= 0;  |
     2 :   channum:= 1;  |
     4 :   channum:= 2;  |
     8 :   channum:= 3;
   ELSE
     RETURN (badChannelSelected);
   END;
   unit[channum]:= iob^.request.unit;
   key[channum]:= iob^.allocKey;
   usertask[channum]:= FindTask(0); (* THIS user task owns it now *)

   RETURN channum;  (* valid channel number (0-3) *)
 END GetChannel;


(* Use IsThatMyChan  to determine if you (still) own a particular channel.
 * The audio device has an arrangement by which a higher priority request
 * for a channel than the one that already owns it can be made.  The higher
 * priority request can actually cause a channel to be stolen from a user.
 * This feature may be implemented in a future version of audiotools,
 * (shared library version), in which, depending on the task's running
 * priority itself, a higher priority task could succeed at GetChannel
 * for a channel that is already owned by another task.
 *)

PROCEDURE IsThatMyChan(channel: LONGINT): LONGINT;
 BEGIN
  IF ((channel < 0) OR (channel > maxChan-1)) THEN RETURN (badChannelSelected);
  ELSIF (usertask[channel] # FindTask(0)) THEN RETURN (notYourChannel); END;
  RETURN 0;     (* if YOU still own the channel *)
END IsThatMyChan;


(* ----------------- internal procedure use only -------------------- *)

PROCEDURE ControlChannel(channel: LONGINT; command: CARDINAL): LONGINT;
 VAR
   error: LONGINT;
   iob: ExtIOBPtr;
   controlIOB: ExtIOB;
 BEGIN
   error:= IsThatMyChan(channel);
   IF (error # 0) THEN RETURN error; END;

   iob:= ADR(controlIOB);
   iob^.request.device:= device;
   iob^.request.message.replyPort:= controlPort;
   iob^.request.unit:= unit[channel];
   iob^.allocKey:= key[channel];

   iob^.request.command:= command;  (* CMD_xxxxx *)
   IF (command = free) THEN
     iob^.request.flags:= noWait + quick; (* ADIOF_NOWAIT |  IOF_QUICK *)
   ELSE
     iob^.request.flags:= quick;       (* IOF_QUICK *)
   END;
   BeginIO(iob);
   WaitIO(iob);  (* returns error in io_Error field; should be 0 *)
   error:= LONGINT(iob^.request.error); (* ADIOERR_NOALLOCATION -10 *)
   RETURN error;
 END ControlChannel;

(* ---------------- USER support procedures -------------------- *)

PROCEDURE StartChannel(channel: LONGINT): LONGINT;
 BEGIN
  RETURN ControlChannel(channel, start);
 END StartChannel;

PROCEDURE StopChannel(channel: LONGINT): LONGINT;
 BEGIN
  RETURN ControlChannel(channel, stop);
 END StopChannel;

PROCEDURE ResetChannel(channel: LONGINT): LONGINT;
 BEGIN
  RETURN ControlChannel(channel, reset);
 END ResetChannel;

PROCEDURE FlushChannel(channel: LONGINT): LONGINT;
 BEGIN
  RETURN ControlChannel(channel, flush);
 END FlushChannel;

PROCEDURE FreeChannel(channel: LONGINT): LONGINT;
```

```
 VAR
   error: LONGINT;
 BEGIN
   error:= ControlChannel(channel, free);
   IF (error # 0) THEN RETURN error; END;
   usertask[channel]:= NIL;   (* free again... *)
   RETURN 0;  (* everything o.k *)
 END FreeChannel;


(* CheckIfDone  to see if everything is finished BEFORE calling FinishAudio *)

PROCEDURE CheckIfDone(): BOOLEAN;
 BEGIN
   RETURN CheckIOBDone();
 END CheckIfDone;

(* Set Period and Volume of a note that is playing.  *)

PROCEDURE SetPV(channel: LONGINT; period, volume: CARDINAL): LONGINT;
 VAR
   error: LONGINT;
   iob: ExtIOBPtr;
   controlIOB: ExtIOB;
 BEGIN
   error:= IsThatMyChan(channel);
   IF (error # 0) THEN RETURN error; END;

   iob:= ADR(controlIOB);
   iob^.request.device:= device;
   iob^.request.message.replyPort:= controlPort;
   iob^.request.unit:= unit[channel];
   iob^.allocKey:= key[channel];

   iob^.period:= period; (* new period *)
   iob^.volume:= volume; (* new volume *)

   iob^.request.command:= perVol;  (* ADCMD_PERVOL *)
   iob^.request.flags:= quick + pervol; (* IOF_QUICK | ADIOF_PERVOL *)
   BeginIO(iob);    (* This one will be synchronous;  *)
     (* affects whatever is playing on this channel at this time. *)
   WaitIO(iob);   (* OK to wait, since it will return *)
   error:= LONGINT(iob^.request.error);   (* ADIOERR_NOALLOCATION -10 *)
   RETURN error;  (* error in io_Error field; should be 0 *)
END SetPV;

(* SetWave creates CHIP RAM, if neccassary (only once per channel)
 * and copies to CHIP RAM (with expand wave) users ARRAY [0..255] OF BYTE,
 * where each element in ARRAY must be in the range -128 to 127  since
 * audio DMA retrieves one word (16 bits) at a time and reads two bytes
 *)

PROCEDURE SetWave(channel: LONGINT;
              VAR waveform: ARRAY OF BYTE): LONGINT;
 VAR
   error: LONGINT;
   i, j, rate: CARDINAL;
   tmptr: ADDRESS;  (* where ADDRESS = POINTER TO BYTE *)
 BEGIN
   error:= IsThatMyChan(channel);
   IF (error # 0) THEN RETURN error; END;

   IF (chipaudio[channel] # 0) THEN  (* not first time *)
     IF (datalength[channel] # waveSize) THEN (* must be sample *)
       FreeMem(chipaudio[channel], datalength[channel]);
       chipaudio[channel]:= 0; datalength[channel]:= 0;
     END;
   END;
   IF (chipaudio[channel] = 0) THEN   (* only allocate if neccessay! *)
     chipaudio[channel]:= AllocMem(waveSize, MemReqSet{chip, memClear});
     IF (chipaudio[channel] = 0) THEN RETURN (outOfMemory); END;
     datalength[channel]:= waveSize; (* for use by FreeMem *)
   END;
       (* ok so far, now copy array to CHIP RAM (with expand wave)   *)
   tmptr:= chipaudio[channel];
   rate:= 1;
   FOR i:= 0 TO 8 BY 1 DO
     j:= 0;
     REPEAT   (* replicate waves in decreasing sample sizes *)
       tmptr^:= waveform[j]; INC(tmptr);  (* increment address *)
       j:= j + rate;
     UNTIL j > 255;
     rate:= rate * 2;
   END;
 RETURN 0;  (* O.K. *)
END SetWave;


(* SetSamp creates CHIP RAM, if neccassary (only once per channel)
 * unless "length"= 0 which just frees up existing sample CHIP RAM or...
 * copies byte by byte from users supplied "sampleaudio" to CHIP RAM,
 * unless "sampleaudio"= 0 which just creates CHIP RAM (without copying)
 * and returns new "sampleaudio" to user, (useful if samples loaded from disk)
 * Note each element in "sampleaudio" must be in the range -128 to 127 since
 * audio DMA retrieves one word (16 bits) at a time and reads two bytes
 *)

PROCEDURE SetSamp(channel: LONGINT;
              VAR sampleaudio: ADDRESS; (* returns new address *)
                  length: LONGINT): LONGINT;
 VAR
   error: LONGINT;
   j: LONGINT;
   chiptr, samptr: ADDRESS;  (* where ADDRESS = POINTER TO BYTE *)
 BEGIN
   error:= IsThatMyChan(channel);
   IF (error # 0) THEN RETURN error; END;

   IF (chipaudio[channel] # 0) THEN   (* free up old mem *)
     FreeMem(chipaudio[channel], datalength[channel]);
     chipaudio[channel]:= 0; datalength[channel]:= 0;
   END;
   IF (length = 0) THEN RETURN 0; END;  (* just free up old mem *)
   IF (length > 131072) THEN length:= 131072; END; (* limit length *)

   IF (chipaudio[channel] = 0) THEN   (* only allocate once per channel! *)
     chipaudio[channel]:= AllocMem(length, MemReqSet{chip, memClear});
     IF (chipaudio[channel] = 0) THEN RETURN (outOfMemory); END;
     datalength[channel]:= length;  (* for use by FreeMem *)
   END;
   IF (sampleaudio = 0) THEN sampleaudio:= chipaudio[channel]; RETURN 0; END;

   (* ok so far, length and sampleaudio nonzero, now copy to CHIP RAM   *)
   chiptr:= chipaudio[channel];
   samptr:= sampleaudio;
   FOR j:= 1 TO length BY 1 DO
       chiptr^:= samptr^; INC(chiptr); INC(samptr);  (* increment address *)
   END;
  RETURN 0;  (* O.K. *)
END SetSamp;


(* -------------- internal procedure use only ------------------*)

PROCEDURE PlayXXXX(channel: LONGINT;
                   wfptr:   ADDRESS;  (* pointer to waveform in CHIP RAM *)
                   len:     LONGCARD;
                   per:     CARDINAL;
                   vol:     CARDINAL;
                   cycles:  CARDINAL;
                   priority:   Byte;
                   messageport: MsgPortPtr;
                   id: LONGINT):   LONGINT;
 VAR
   error: LONGINT;
   iob: ExtIOBPtr;
 BEGIN
   iob:= GetIOB(channel);
   IF (iob # NIL) THEN  (* set the parameters *)
      iob^.request.unit:= unit[channel];
      iob^.allocKey:= key[channel];
      iob^.data:= wfptr;
      iob^.length:= len;
      iob^.period:= per;
      iob^.volume:= vol;
      iob^.cycles:= cycles;
      iob^.request.message.node.pri:= priority;
      iob^.identifier:= id;    (* for support of tell-me-when-note-starts *)
```

```
      iob^.request.command:= write;  (* CMD_WRITE *)
      iob^.request.flags:= pervol;  (* ADIOF_PERVOL *)

      (* Initialize message port.  If NIL, then no pushing back of a message.
       * If nonzero, message gets recirculated by ReEmployIOB until
       * the user finally acknowledges it by using MayGetNote. *)

       iob^.writeMsg.replyPort:= messageport;
       IF (messageport # NIL) THEN
          (*  "reply" to this message -  ADIOF_WRITEMESSAGE *)
     iob^.request.flags:= iob^.request.flags + writeMessage;
        END;
        BeginIO(iob);
        RETURN 0;      (* all went ok *)
   END;
  RETURN (outOfMemory);  (* (else-part) iob was zero, couldn't do the above *)
 END PlayXXXX;

(* PlayNote - starts a sound on the channel with specified period and volume.
 * This nice little routine takes a note and plays it on the given
 * voice.  The note is basically an integer from
 * 0 to 11 (c to b) plus 12 per octave above the first and lowest,
 * which yields a note range of 0 to 95.
 * The waveform to use is determined by adding an index (woffsets[])
 * dependant on the octave to waveform in chipaudio[channel] as setup by
 * SetWave.  The length of the waveform (in wlen[]) is likewise dependant
 * on the octave.  Note that octaves start with zero, not one.
 * The period and volume can be modified later, using SetPV.
 *)

PROCEDURE PlayNote(channel: LONGINT;   (* specify channel number 0-3 *)
                   note:    CARDINAL;  (* specify note number 0-95 *)
                   volume:   CARDINAL;   (* volume 0-64  *)
                   duration: CARDINAL; (* duration  1000ths of a sec. *)
                   priority:   Byte;     (* force a range -128 to 127 *)
                   messageport: MsgPortPtr;
                   id: LONGINT);
 VAR
   error: LONGINT;
   period, octave: CARDINAL;
   ipart, jpart: CARDINAL;
   length: LONGCARD;
   wavepointer: ADDRESS;     (* where to find start of waveform *)
   cycles: CARDINAL;
 BEGIN
   error:= IsThatMyChan(channel);
   IF (error # 0) THEN RETURN; END;

   IF (note > 95) THEN note:= 95; END;
   IF (volume > 64) THEN volume:=64; END;
   octave:= note DIV 12;

   IF (chipaudio[channel] = 0) THEN RETURN; END; (* no SetWave or SetSamp !!*)
   wavepointer:= chipaudio[channel] + ADDRESS(woffsets[octave]);
   length:= wlen[octave];
   period:= perval[note MOD 12];

   (* divide duration into two parts - ipart & jpart - for calculations *)
   IF (duration > 1000) THEN ipart:= duration DIV 1000; ELSE ipart:=0; END;
   jpart:= duration - (ipart * 1000);

   (* fool it a little so we don't get integer overflow...            *)
   (* 3.5 million times 1000 is about all we can take in a 32 bit word *)

   cycles:= ((LONGCARD(audClock) * ipart)+(LONGCARD(audClock) * jpart) DIV 1000)
            DIV (LONGCARD(length) * period);
   IF ((cycles = 0) AND (duration # 0)) THEN cycles:= 1; END;

  error:= PlayXXXX(channel,wavepointer,length,period,volume,cycles,
                   priority,messageport,id);
  RETURN       (* just ignore error *)
 END PlayNote;


(* PlayFreq - in this version is for scalar values of frequency only.
 * Minimum value is 28Hz, practical maximum is about 7000Hz.
 * Period is calculated from frequency to within 127 to 500, otherwise,
 * if the frequency is out of range of what we have in our wave tables
 * currently, we have to reject the command.
 *)

PROCEDURE PlayFreq(channel: LONGINT;    (* specify channel number 0-3 *)
                   freq:    CARDINAL;   (* specify scalar freq 28-7000 Hz *)
                   volume:   CARDINAL;  (* volume 0-64 *)
                   duration: CARDINAL;  (* 1000ths of a second *)
                   priority:     Byte; (* force a range -128 to 127 *)
                   messageport: MsgPortPtr;  (* for use by MayGetNote *)
                   id: LONGINT);
 VAR
   error: LONGINT;
   period, octave: CARDINAL;
   ipart, jpart: CARDINAL;
   length: LONGCARD;
   wavepointer: ADDRESS;   (* where to find start of waveform *)
   cycles: CARDINAL;
   i: CARDINAL;
   accept: BOOLEAN;
 BEGIN
   error:= IsThatMyChan(channel);
   IF (error # 0) THEN RETURN; END;

   IF (freq = 0) THEN RETURN; END;
   IF (volume > 64) THEN volume:= 64; END;

   i:= 0;  (* see if we CAN represent this frequency, if not, reject it *)
   LOOP                  (* figure out which waveform to use... *)
     octave:= i;         (* start with the first wlen value because *)
     accept:= FALSE;     (* we want to use the longest waveform we can. *)
     period:= LONGCARD(audClock) DIV (LONGCARD(freq) * (wlen[octave]));
     IF (period > 500) THEN EXIT; END; (* freq less than 28Hz. *)
     IF (period > 127) THEN accept:= TRUE; EXIT; END;
     i:=i+1; IF (i > 8) THEN EXIT; END;
   END;
   IF (accept = FALSE) THEN RETURN; END;  (* reject it *)

   IF (chipaudio[channel] = 0) THEN RETURN; END; (* no SetWave or SetSamp !!*)
   wavepointer:= chipaudio[channel] + ADDRESS(woffsets[octave]);
   length:= wlen[octave];

   (* divide duration into two parts - ipart & jpart - for calculations *)
   IF (duration > 1000) THEN ipart:= duration DIV 1000; ELSE ipart:=0; END;
   jpart:= duration - (ipart * 1000);

   (* fool it a little so we don't get integer overflow...            *)
   (* 3.5 million times 1000 is about all we can take in a 32 bit word *)

   cycles:= (LONGCARD(freq) * ipart) + (LONGCARD(freq) * jpart) DIV 1000;
   IF ((cycles = 0) AND (duration # 0)) THEN cycles:= 1; END;

  error:= PlayXXXX(channel,wavepointer,length,period,volume,cycles,
                   priority,messageport,id);
  RETURN       (* just ignore error *)
 END PlayFreq;


(* MayGetNote - is used to synchronize the Play audio routines, using
 * messageport and id, (parameters of the Play routines).
 * where uport is the pointer to the port you received from InitAudio.
 *
 *  when flag = FALSE, the routine returns immediately, with an id = 0
 *  (no id available), or the value of the first id to arrive at the port.
 *
 *  when flag = TRUE, the routine will wait if (and only if) there is no id.
 *  In other words, you can cause your task to go to sleep until the
 *  next note begins to play.  You decide what to do for a specific note.
 *
 *   CAUTION - if there are no more notes with messageport nonzero in
 *         the queue and you specify TRUE for the flag, you can cause your
 *         task to sleep forever!!
 *)

PROCEDURE MayGetNote(uport: MsgPortPtr; flag: BOOLEAN): LONGINT;
 VAR
   aum: auMsgPtr;

 BEGIN
  LOOP
   aum:= auMsgPtr(GetMsg(uport));   (* is a message there? *)
```

```
    IF (aum # NIL) THEN  (* There was a message! *)
       (* The user has seen this msg, so the system can deallocate
        * the iob in which it occurs anytime in the future.
        * Now that we have received the message at our own reply
        * port, it belongs to us and we can do whatever we want
        * to it.  Set the reply port value to zero now, as a signal
        * to FreeIOB that it can really do that!
        *)
       aum^.message.replyPort:= NIL;
       EXIT;  (* from LOOP with message *)
    END;
    IF (flag = TRUE) THEN
       (* let caller sleep while waiting for any identified iob to appear. *)
       WaitPort(uport);  (* Note: WaitPort does NOT remove message from port *)
       flag:= FALSE;
    END;
  END;
 RETURN (aum^.identifier);  (* return the LONG value *)
END MayGetNote;


(* PlaySamp - play a sampled sound:
 *  Identical to PlayFreq but the parameters are interpreted differently.
 *  "freq" now becomes "period" interpreted as sampling_rate,
 *  must be in the range of 127 to 500.
 *  "duration" still is expressed in 1000ths of a second to play it.
 *  (as with the audio device itself, a duration of 0 means do it forever
 *   or until the audio device is reset or the channel is flushed or
 *   until this command is explicitly aborted.)
 *)

PROCEDURE PlaySamp(channel: LONGINT;   (* specify channel number 0-3 *)
                   period:  CARDINAL;   (* period value 127 to 500 *)
                   volume:    CARDINAL; (* volume 0-64 *)
                   duration: CARDINAL;  (* 1000ths of a second *)
                   priority:     Byte;  (* force a range -128 to 127 *)
                   messageport: MsgPortPtr;  (* for use by MayGetNote *)
                   id: LONGINT);
VAR
  error: LONGINT;
  wavepointer: ADDRESS;   (* where to find start of sample to play *)
  cycles: CARDINAL;
  ipart, jpart: CARDINAL;
  length: LONGCARD;
 BEGIN
   error:= IsThatMyChan(channel);
   IF (error # 0) THEN RETURN; END;

   IF (period > 500) THEN period:= 500;   (* Note: or reject it ? *)
   ELSIF (period < 127) THEN period:= 127; END;

   IF (volume > 64) THEN volume:= 64; END;

   IF (chipaudio[channel] = 0) THEN RETURN; END; (* no SetWave or SetSamp !!*)
   wavepointer:= chipaudio[channel];
   length:= datalength[channel]; (* as set by SetSamp *)

   (* divide duration into two parts - ipart & jpart - for calculations *)
   IF (duration > 1000) THEN ipart:= duration DIV 1000; ELSE ipart:=0; END;
   jpart:= duration - (ipart * 1000);

   (* fool it a little so we don't get integer overflow...          *)
   (* 3.5 million times 1000 is about all we can take in a 32 bit word *)

   cycles:= ((LONGCARD(audClock) * ipart)+(LONGCARD(audClock) * jpart) DIV 1000)
            DIV (LONGCARD(length) * period);
   IF ((cycles = 0) AND (duration # 0)) THEN cycles:= 1; END;

  error:= PlayXXXX(channel,wavepointer,length,period,volume,cycles,
                   priority,messageport,id);
  RETURN        (* just ignore error *)
 END PlaySamp;


(* If the user says FinishAudio, IT MEANS FINISH AUDIO.
 * Flush anything that is still in play, NOW.  You can
 * use "CheckIfDone()" to see if everything is finished
 * BEFORE you call FinishAudio.  If CheckIfDone() is
 *  (FALSE), it means that something is still playing.
 *)

PROCEDURE FinishAudio(uport: MsgPortPtr);
 VAR
   error: LONGINT;
   aum: auMsgPtr;     (* A little bigger than a standard message, *)
   i: LONGINT;        (* but this routine will not really know    *)
                      (* (or care) about the difference. *)
 BEGIN
   IF (uport # NIL) THEN
      FOR i:=0 TO maxChan-1 BY 1 DO
         error:= FlushChannel(i);   (* error is dummy function return *)
      END;

      WHILE (CheckIOBDone() = FALSE) DO
         Delay(12);   (* Be a good multitasking neighbor: sleep a little *)
      END;

      aum:= auMsgPtr(GetMsg(uport));  (* prepare to empty the port *)
      WHILE (aum # NIL) DO
         aum^.message.replyPort:= NIL;       (* let system deallocate it *)
         aum:= auMsgPtr(GetMsg(uport));
      END;

      ReEmployIOB();   (* free all static and dynamic messages *)

      FOR i:=0 TO maxChan-1 BY 1 DO
        error:= FreeChannel(i);   (* error is dummy function return *)
      END;
      DeletePort(uport);
   END;

   IF (device # NIL) THEN CloseDevice(ADR(openIOB)); END;
   FOR i:=0 TO maxChan-1 BY 1 DO
     IF (chipaudio[i] # 0) THEN
        FreeMem(chipaudio[i], datalength[i]);
        chipaudio[i]:= 0; datalength[i]:= 0;
     END;
     IF (replyPort[i] # NIL) THEN DeletePort(replyPort[i]); END;
   END;
   IF (controlPort # NIL) THEN DeletePort(controlPort); END;

 END FinishAudio;

END AudioTools.imp


MODULE AudioDemo;       (* test of the AudioTool library module routines *)
                        (* adapted to M2Amiga Modula-2 by Anthony Bryant *)
FROM SYSTEM IMPORT
 ADR,ADDRESS,BYTE,FFP;

FROM AudioTools IMPORT
 InitAudio, GetChannel, StopChannel, StartChannel,
 SetPV, SetWave, SetSamp, PlayNote, PlayFreq,
 MayGetNote, PlaySamp, CheckIfDone, FinishAudio;

FROM Arts IMPORT  TermProcedure;

FROM Dos IMPORT   Delay;

FROM Exec IMPORT  Byte, UByte, MsgPortPtr;

FROM RandomNumber IMPORT  RND;

FROM Terminal IMPORT  WriteLn, WriteString;


TYPE
  waveArray = ARRAY [0..255] OF BYTE;
CONST
  noiselen = 1024;
VAR
   i: CARDINAL;
   channel, error, myid: LONGINT;
   myport:  MsgPortPtr;
   sawtooth: waveArray;
   triangle: waveArray;
```

```
    square:   waveArray;
    sinewave: waveArray;
    noise:   ARRAY [0..noiselen-1] OF BYTE;
    noiseptr: ADDRESS;

PROCEDURE MakeWaveArrays;
 VAR
   i: INTEGER;  (* index to waveArray *)
   trv,sqv: Byte;
   sine,cosine,step: FFP; (* Fast Floating Point *)
 BEGIN
   trv:=0; sqv:=127;
   sine:=0.0; cosine:=1.0; step:=0.0247;
   FOR i:=0 TO 255 BY 1 DO
     sawtooth[i]:= Byte(127 - i);
     triangle[i]:= Byte(trv);
     square[i]:=   Byte(sqv);
     sinewave[i]:= Byte(INTEGER(127.0 * sine));
     IF (i>62) AND (i<190) THEN DEC(trv,2); ELSE INC(trv,2); END;
     IF (i>127) THEN sqv:=-128; END;
     cosine:= cosine - (step * sine);
     sine:= sine + (step * cosine);
   END;
 END MakeWaveArrays;

PROCEDURE MakeNoiseArray;
 VAR
   i: CARDINAL;
 BEGIN
   FOR i:=0 TO noiselen-1 BY 1 DO
     (* Use a PseudoRandom number generator to get white noise, well almost *)
     (* this really could do with a larger array, but what the heck *)
     noise[i]:= Byte(127 - RND(256));
   END;
   noiseptr:= ADR(noise[0]);
 END MakeNoiseArray;

PROCEDURE Cleanup;
BEGIN
   IF (myport # NIL) THEN
     FinishAudio(myport);
     WriteString("Done!");
     WriteLn;
   END;
END Cleanup;

BEGIN  (* AudioDemo *)
   TermProcedure(Cleanup);  (* don't leave an untidy Amiga *)
   WriteLn;
   WriteString("Demo of AudioTools. ");
   WriteLn;
   MakeWaveArrays;   MakeNoiseArray;
   WriteLn;
   myport:= InitAudio();  (* save for MayGetNote and FinishAudio *)
   IF (myport = NIL) THEN WriteString("Problem in InitAudio!"); RETURN; END;

   FOR i:= 0 TO 3 BY 1 DO
     channel:= GetChannel(-1);  (* any free channel ? *)
     IF (channel = -1) THEN WriteString("Cannot get a channel!"); RETURN; END;
     error:= StopChannel(channel);
     IF (error # 0) THEN WriteString("Error in stopping channel!"); RETURN; END;
   END;
     WriteString("Got & Stopped 4 channels O.K.");
     WriteLn;

   (*  assign waveArrays using SetWave()  *)
       (* SetWave(channel, wavearray) *)
   error:= SetWave(0, sawtooth);
   IF (error # 0) THEN WriteString("Error in SetWave, ch 0"); RETURN; END;
   error:= SetWave(1, triangle);
   IF (error # 0) THEN WriteString("Error in SetWave, ch 1"); RETURN; END;
   error:= SetWave(2, square);
   IF (error # 0) THEN WriteString("Error in SetWave, ch 2"); RETURN; END;
   error:= SetWave(3, sinewave);
   IF (error # 0) THEN WriteString("Error in SetWave, ch 3"); RETURN; END;
   WriteString("SetWave assigned all 4 waveforms.");
   WriteLn;

   (* PlayXXXX(channel, note/freq,  vol, duration, priority,messageport, id) *)
   FOR i:= 0 TO 48 BY 1 DO            (* all notes, 1/8 sec. *)
     PlayFreq(0, 10*i,     32, 125, 0, NIL, 0);
     PlayNote(1, 95-i,     32, 125, 0, NIL, 0);
   END;
   WriteString("Queued up Lo-Freq PlayNotes and PlayFreqs ");
   WriteLn;

   (* SMACK IN THE MIDDLE HERE, INSTALL TEST OF MESSAGING *)
   (* Make the note LOUDER so we can tell for sure *)

   PlayNote(0, 3,          63, 500, 0, myport, 7);
   WriteString("Queued up Test of Messaging with id = 7");
   WriteLn;
   FOR i:= 50 TO 95 BY 1 DO          (* all notes, 1/8 sec. *)
     PlayFreq(0, 10*i,     32, 125, 0, NIL, 0);
     PlayNote(1, 95-i,     32, 125, 0, NIL, 0);
   END;
   WriteString("Queued up Hi-Freq PlayNotes and PlayFreqs ");
   WriteLn;

   FOR i:= 0 TO 3 BY 1 DO
     error:= StartChannel(i);
     IF (error # 0) THEN WriteString("Error starting channel!"); RETURN; END;
   END;
   WriteString("Started all channels O.K.");
   WriteLn; WriteLn;

   WriteString("SLEEP now - the prioritized myid will wake me up.");
   WriteLn;
   myid:= MayGetNote(myport, TRUE); (* FALSE means don't sleep *)
   WriteLn;
   WriteString("Hey!  Note I identified as ");
   IF (myid = 7) THEN WriteString("7"); ELSE WriteString("0"); END;
   WriteString(" just started to play");
   WriteLn; WriteLn;

   WriteString("Wait 6 seconds ... ");
   Delay(300);
   WriteString(" CheckIfDone ? - ");
   IF (CheckIfDone() = FALSE) THEN WriteString("FALSE");
   ELSE WriteString("TRUE"); END;
   WriteLn; WriteLn;
   WriteString("Now Queue up PlayNotes with duration > 2000.");
   WriteLn;
   PlayNote(0, 23,    32, 2000, 0, NIL, 0);
   PlayNote(1, 27,    32, 2300, 0, NIL, 0);
   PlayNote(2, 30,    32, 2600, 0, NIL, 0);
   PlayNote(3, 35,    32, 2900, 0, NIL, 0);
   WriteLn;
   Delay(150); (* Waits 3 seconds after letting the last note begin
       * (because last note is 2900/1000ths long).
       *)
   WriteString("Last note should have ended... ");
   WriteString("CheckIfDone ? - ");
   IF  CheckIfDone()  THEN WriteString("TRUE");
   ELSE WriteString("FALSE"); END;
   WriteLn; WriteLn;

   (* assign noise array to ch 0, using SetSamp() *)
       (* SetSamp(channel, sampleaudio, length) *)
   error:= SetSamp(0, noiseptr, noiselen);
   IF (error # 0) THEN WriteString("Error in SetSamp, ch.0"); RETURN; END;
   WriteString("Finally, some noise, using PlaySamp");
   WriteLn;
   PlaySamp(0,    327, 48, 6000, 0, NIL, 0);
   (* PlaySamp(chan, samplrate, volume, 6.0sec duration, pri, msgpt, id)   *)

   FOR i:= 6 TO 1 BY -1 DO   (* try a 6 second decay of volume *)
     Delay(50);
     error:= SetPV(0,327, i*8);
     IF (error # 0) THEN WriteString("Error in SetPV, ch.0"); RETURN; END;
   END;
   (* If you take out this delay, you'll see that FinishAudio means
    * FINISH AUDIO... it cuts off the notes right in the middle if necessary! *)
   WriteString("Noise should have ended.");
   WriteLn; WriteLn;
   WriteString("Ready to Finish Audio.");  (* via TermProcedure()  *)
   WriteLn;
   RETURN;
END AudioDemo.mod
```

Reading C Declarations

A guide for the mystified

by Eric Giguère

Eric Giguère has been involved with the design of an ANSI-conformant C compiler at the Computer Systems Group at the University of Waterloo, where he is a computer science student in the co-op program.

One of the most confusing aspects of the C programming language is the way in which variable types are declared. These type declarations manage to mystify even experienced programmers. This guide is intended to help in deciphering those declarations, including the new twists introduced by the ANSI Standard. Several examples are given to help illustrate the methods, and a short quiz at the end allows you to test yourself on a variety of sample declarations.

What is a declaration?

Throughout this guide the word *declaration* will be used to refer to a variable declaration statement. The following two statements are both declarations:

```
static int x;
register short y = 0;
```

We will,however, ignore any initialization and concentate only on the part of the statement to the left of the '=' operator.

Pre-ANSI declarations

A C type declaration can be broken into two distinct parts: a *type specifier* and a *declarator*. The type specifier is the easy part: it declares the (optional) storage class (either **auto, static, register,** or **extern**) and a type (such as **int** or **unsigned char**). The declarator is the one that causes the confusion; it specifies whether a variable is to be considered a simple type, a pointer, an array, a function, or a combination of these. The *identifier*, the variable name, is found in the middle of the declarator.

The rules we'll use will allow you to follow a declaration by writing a phrase on a piece of paper at each step in the decoding process. The final result will be a phrase of the form "**declare** *variable-name* **as** ..." stating in plain English what exactly is being declared.

Rules

The rules for reading a type declaration can be summarized as follows:

1) Find the identifier. This is your starting point. On a piece of paper, write "declare identifier as".

2) Look to the right. If there is nothing there, or there is a right parenthesis ')', go to step 4.

3) You are now positioned on either an array (left square bracket) or a function (left parenthesis) *descriptor*. There may be a sequence of these, ending either with an unmatched right parenthesis or the end of the declarator (a semicolon or an '=' for initialization). For each such descriptor, reading from left to right:

- if an empty array '[]', write "array of"
- if an array with a size, write "array *size* of"
- if a function '()', write "function returning"

Stop at the unmatched parenthesis or the end of the declarator, whichever comes first.

4) Returning to the starting position, look to your left. If there is nothing there, or there is a left parenthesis '(', go to step 6.

5) You are now positioned on a pointer descriptor, '*'. There may be a sequence of these to the left, ending either with an unmatched left parenthesis '(' or the start of the declarator. Reading from *right to left*, for each pointer descriptor write "pointer to". Stop at the left parenthesis or the start of the declarator, whichever is first.

"It may sound confusing, but after a while it becomes second nature..."

6) At this point you have either a parenthesized expression or the complete declarator. If you have a parenthesized expression, consider it as your new starting point and return to step 2.

7) Write down the type specifier. Stop.

It may sound confusing, but after a while it becomes second nature. To illustrate, we shall read a few sample C declarations using this method.

Example 1: **static int *x;**

Our starting point is the identifier x. We write "declare x as". We look to the right. Nothing there, go to step 4. We have a pointer descriptor, so we go to step 5. We write "pointer to" and stop, since we've reached the start of the declarator. At step 6 we see we have the complete declarator, so we move on to step 7, write "static int" and stop.

The declaration then reads: "declare x as pointer to static int".

Example 2: **char *argv[];**

Step 1, write "declare argv as". Step 2, array to the right. Step 3, write "array of". Step 4, pointer to the left. Step 5, write "pointer to". Step 6, complete declaration. Step 7, write "char". Stop.

The declaration is: "declare argv as array of pointer to char". Note that it's *not* a pointer to an array of char. Arrays have *precedence* over pointer descriptors and are read first.

Example 3: **int (*ptar)[10];**

Step 1, write "declare ptar as". Step 2, parenthesis on the right, go to step 4. Step 5, write "pointer to". Step 6, parenthesized expression, go to step 2. Step 3, write "array 10 of". Step 4, start of declarator on the left, go to step 6. Step 7, write "int". Stop.

The declaration is: "declare ptar as pointer to array 10 of int". Here the parentheses were used to force the pointer descriptor to have precedence over the array descriptor. The array has ten elements.

Example 4: **int (*fp)();**

Step 1, write "declare fp as". Step 2, parenthesis on the right, go to step 4. Step 5, write "pointer to". Step 6, parenthesized expression, go to step 2. Step 3, write "function returning". Step 4, start of declarator on the left, go to step 6. Step 7, write "int". Stop.

The declaration is: "declare fp as pointer to function returning int". Function and array descriptors have the same precedence, so again the parentheses were necessary to force the the pointer descriptor to be read first. (Note then that **int *fp();** declares **fp** as a **function returning pointer to int**.)

Example 5: **int *(*list[MAX])();**

Step 1, the identifier is **list** (MAX is the array size), write "declare list as". Step 2, array on the right. Step 3, write "array MAX of". Step 4, pointer on the left. Step 5, write "pointer to". Step 6, complete declarator. Step 7, write "int". Stop.

The declaration is: "declare list as array MAX of pointer to function returning pointer to int". Without the parentheses, we would have had an array of functions returning pointers to pointers to **int**, which is illegal in C because arrays cannot hold functions.

Example 6: **char *table[10][20]**

Step 1, write "declare table as". Step 2, array on right. Step 3, write "array 10 of" followed by "array 20 of". Step 4, pointer on the left. Step 5, write "pointer to". Step 6, complete declaration. Step 7, write "char". Stop.

The declaration is: "declare table as array 10 of array 20 of pointer to char".

The important thing to remember while reading these declarations is that functions and arrays are read left-to right and have precedence over pointers, which are read right-to-left.

ANSI additions

The draft proposed ANSI Standard for C makes a few additons to what a C type declaration can look like. The basic structure of *type specifier* and *declarator* remains unchanged. However, a function designator may now include both the number and the types of parameters the function accepts and pointers may also be modified by the keywords **const** and **volatile**.

A further addition is the *abstract declarator*, used to describe function parameters. An abstract declarator is a declarator without an identifier. Thus instead of ***a[]** you could conceivably use *[]. This makes it a bit harder to spot the correct starting position in the declaration. Note that abstract declara-

tors can only be used in certain situations - an identifier is a necessary part of most declarations.

The rules given in the previous section need to be amended slightly to handle these new additions, but rather than list them we shall go through a few more examples.

Example 7: **int func(char *, int);**

Step 1, find the identifier, write "declare func as". Step 2, notice the function designator to the right; go to step 3. Here we make some changes. Write simply "function". Now shift your attention *inside* the parentheses making up the function descriptor. These are the types of parameters that the function expects. Each parameter is a declaration in itself, with a type specifier and an abstract declarator. Put aside the current declaration for now, write "(with first parameter of type", and go to step 2 with **char *** as the new starting point:

> Step 2, nothing on the right, go to step 4. Step 5, write "pointer to". Step 6, end of declaration. Step 7, write "char". stop.

Now go on to the second parameter. Write "and with second parameter of type", and go to step 2 with **int** as the new starting point:

> Step 2, nothing on the left. Step 4, nothing on the right. Step 6, end of declaration. Step 7, write "int". Stop.

There are no more parameters to be dealt with so we continue with our original declaration from where we left off in step 3, first writing ") returning". Step 4, no pointers on the left. Step 6, end of declaration. Step 7, write "int". Stop.

The complete declaration is: "declare func as function (with first parameter of type pointer to char and with second parameter of type int) returning int".

You can (and probably should) ignore the parameters. This would then give us: "declare func as function returning int". All we are doing when declaring the parameters is adding more detail.

Example 8: **const int *ptrl;**

This case is handled using the old rules to give us: "declare ptrl as a pointer to const int", where **const** specifies that the **int** cannot be modified.

Example 9: **int *const ptr2;**

This is an example of the new syntax involving **const** and **volatile**. Either or both of these two types may be placed *after* a pointer designator to assign the attributes to the pointer itself. Back to the rules: Step 1, start at **ptr2**, the only choice for identifier (since **const** and **volatile** are reserved words in ANSI C); write "declare ptr2 as". Step 2, nothing to the right, go to step 4. Step 4, we have a '*' followed by **const**. This qualifies the pointer descriptor. Step 5, write "const pointer to". Step 6, end of declaration. Step 7, write "int". Stop.

The declaration is: "declare ptr2 as a const pointer to int". The distinction to make between this declaration and the one in example 8 is that the **const** attribute is associated with the *pointer*, and not the type specifier. So while the pointer itself cannot be modified, the type it points to can.

The **volatile** type can also be used in this fashion, either by itself or in combination with **const**. Write it the same way.

Pointer descriptors do not require these attributes, however, so a declaration of the form **int **a** is still entirely valid.

Typing overload

Obviously, the more information included in a declaration, the harder it gets to read.

Example 10: **extern char *const (*goop(char *b))(int,long);**

Where do we start? There are two identifiers in this declaration, **goop** and **b**. Notice, however, that **char *b** is declaring a function parameter; **goop** is our starting point. Step 1, write "declare goop as". Step 2, function on the right. Step 3, write "function". Now look inside the function to find any parameters. Write "(with one parameter of type char *". Notice how we ignore the identifier **b**. Function parameters declared in function prototypes can be declared with abstract or normal declarators; in the latter case, the identifier is ignored. Only the typing information counts. (The identifiers are *not* ignored in ANSI function *definitions*, but we're declaring a prototype here.)

Now go back to step 3, complete the function and write ") returning". Step 4, a pointer to the left of **goop**. Step 5, write "pointer to". Step 6, a parenthesized expression, back to step 2. Step 2, function on the right. Step 3, write "function". Now go inside, write "(with first parameter of type int and with second parameter of type long". End of function, back to step 3 and write ") returning". Step 4, pointer (with **const**) on the left. Step 5, write "const pointer to". Step 6, end of declaration. Step 7, write "extern char". Stop.

The declaration is (take a deep breath): "declare goop as function (with one parameter of type char *) returning pointer to function (with first parameter of type int and second parameter of type long) returning const pointer to extern char". (Whew!) Or, if you choose to ignore the parameters: "declare goop as function returning pointer to function returning const pointer to extern char", which is a bit less intimidating.

Parentheses

One of the harder things to do is to decide what a given set of parentheses implies - are we declaring a function or are we simply grouping an expression? The simplest way to differentiate the two is to look at the placement of the type specifiers. If a type specifier immediately follows a left parenthesis, that parenthesis is the start of a function descriptor and the type is part of a function parameter. Empty parentheses also signify a function. Otherwise the parentheses are grouping (perhaps unnecessarily - but it's better to have too many than too few) the expression.

The Quiz

After all this work, have we accomplished anything? You might think that working out those examples step-by-step was overkill. Maybe. But read them over carefully once or twice, work through the following examples, and you won't ever be confused by a C declaration again. Answers can be found below.

The best way to familiarize yourself with C declarations is to work through a few of them on your own. Here are a few for you to try and decipher. (Answers below).

(a) **char ****q[30];**
(b) **char **(**q)[30];**
(c) **extern int (x)[ ];**
(d) **long (*a [ ])(char, char);**
(e) **int *(*(*(*b)( ))[10])( );**
(f) **char *strprt(char(*)(int),unsigned char);**
(g) **int (*const ab[ ])(unsigned int);**

The Answers

(a) declare q as array 30 of pointer to pointer to pointer to pointer to char
(b) declare q as pointer to pointer to array 30 of pointer to pointer to char
(c) declare x as array of extern int
(d) declare a as array of pointer to function (with two parameters of type char) returning long
(e) declare b as pointer to function returning pointer to array 10 of pointer to function returning pointer to int
(f) declare strprt as function (with first parameter of type pointer to function (with parameter of type int) returning char and with second parameter of type unsigned char) returning pointer to char
(g) declare ab as array of const pointer to function (with parameter of type unsigned int) returning int

Amiga

500 Computer	
501 512K Expansion	$1000
1084SColor Monitor	
1010 3½" external	$159.00
1020 5¼" external	$189.00
2000 Computer	$1489.00
2010 3½" internal	$160.00
2052 2MB Expansion	$399.00
2088 Bridgeboard w/5¼" drive	$499.00

Epson

LX-800	$199.00
FX-86e	$299.00
FX-850	$359.00
FX-1050	$509.00
EX-800	$439.00
LQ-500	$338.00
LQ-850	$539.00
LQ-1050	$759.00
LQ-2500	$899.00

Panasonic

KX-P1080i	$159.00	KX-P1592	$399.00
KX-P1091i	$199.00	KX-P1595	$469.00
KX-P1092i	$319.00	KX-P3131	$310.00
KX-P1524	$539.00	KX-P3151	$479.00

Packard Bell

1200B $79.00 2400B $149.00
Hayes compatible external modem

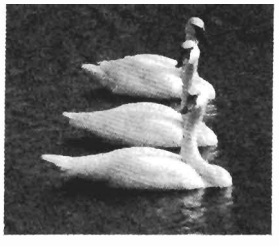
MasterCard **PC Plus** VISA
918-337-0266
Everything is in stock, ship the same day
Amiga is the registered trademark of Commodore Business Machines
Authorized Dealer for Amiga, Epson, Panasonic

CygnusEd Professional

by CygnusSoft Software. Published by ASDG Inc.

"The New King of Editors,"
68000er, February 1988

"This editor is definitely one of the best I've used on any machine, and I've used many."
The Amiga Sentry, Sept. 1988



CygnusEd Professional brings unparalleled levels of flexibility and speed to the **Amiga.** features include:

Crash Recovery. Variable screen resolution up to 1000 by 800. Full Macro capability. Over 200 functions and variables available through AREXX. Hot Key operation. Screen refresh at over 30,000 cps. Search and replace at over 100,000 cps. Blitter based horizontal and vertical scrolling. Vertical and Horizontal block cut and paste. Many Word Processing features. Many Programmer features. Edit multiple files on screen at same time.

Asynchronous Printer support. Super interprocess communications facilities. DOS and AREXX commands can be launched from inside the editor. User definable "bookmarks". International keyboard and video support. Mouse or keyboard based Turbo Scrolling. Keyboard and mouse can be interchanged for everything except entering text. Will create icons for WorkBench users. Edits binary files. And **MUCH more!**

ASDG INCORPORATED
925 Stewart Street
Madison, WI 53713
(608) 273-6585

Available At Your Local Dealer **$99.95** U.S. Dollars

CygnusEd Professional Copyright 1988 By CygnusSoft Software. Amiga and Workbench are trademarks of Commodore-Amiga Inc.
Developers are encouraged to contact ASDG for possible publication of your products.

Some examples of shell programming

A detailed look at Execute's scripts

by John Faichney

John Faichney is with the Arts Television Centre in Toronto. In this article, John demonstrates how you can more fully exploit the functionality of the tools supplied with Workbench 1.3.

This article features examples of shell programs - files consisting of statements that, upon execution of the file, run normal AmigaDOS commands. The point deserves emphasis: none of the programs featured in this article contains any 'object code', though a few employ console control sequences. Moreover, none contains any non-standard AmigaDOS v1.3 commands, even in those cases where their use would have made the programs easier to write.

Several examples address typical problems of file and disk management, and may be of practical interest to users for whom address-space and/or file-system memory are scarce resources - those to whom the overhead of 'object code' approaches to file and disk management (DirUtil, DiskMan, and the like) is unattractive. Other examples, while fully functional, will be of interest primarily as demonstrations of the programming techniques involved. Yet others are nothing less than outstanding examples of shell programming.

1. An outstanding example of shell programming

Enter the following at the shell prompt:

```
1.RAM DISK:> Join * as Prompt.test
```

Now enter the following (and press RETURN):

```
Prompt "ESC[31m%n.%s>ESC[32m "
```

(Note that the ESC key doesn't print to the screen as a visible character). Type the CTRL-\ key combination to send end-of-file to the Join command. Now enter:

```
1.RAM DISK:> Execute Prompt.test
```

The effect of this will be to render the shell window and prompt in Workbench colour 1, and command line I/O in colour 2.

2. Creating and displaying text files with Join and Echo

As demonstrated in the previous section, by passing the 'current window' argument (an asterisk by itself) to Join as its 'input file' parameter, we can create text files like *Prompt.test* directly from the shell. Specifying the current window as Join's 'AS' (destination file) parameter provides a substitute for the Type command:

```
1.RAM DISK:> Join Prompt.test as *
```

While Join does not require more than one argument to its input file parameter, we can also use Join to concatenate several instances of the current window:

```
1.RAM DISK:> Join * * * * * * * * * * as foo
```

Granted, this is hardly the most user-friendly way to create a text file. Consider, however, the following:

```
1.RAM DISK:> Join * Prompt.test as S:StandOut
```

Now enter:

```
; An outstanding example of shell programming
```

and type CTRL-\. This time we'll use Type to display the file:

```
1.RAM DISK:> Type S:StandOut number
```

We get:

```
1 ; An outstanding example of shell programming
2 Prompt "%n.%s> "
```

As these examples show, commands that normally direct their I/O streams from/to the file system can easily be made to orient to other types of device.

"So what?" you may say. "Surely shell programming has more going for it than a few tricks for creating simple text files?" Ah, but a few tricks for creating simple script files will

come in handy any time we need to test some combination of conditions or parameters and don't need the features of a full editor. But while it doesn't really matter how scripts get written, a few observations on script presentation are in order:

• All scripts will be presented here with line numbers and, where appropriate, line breaks; these features are *not* integral to the scripts.

• Both the semicolon command line delimiter - ';' - and the dot-&-space directive - '. ' - will be used to distinguish in-code comments.

Note, however, that as of v1.2, at least, not all commands will tolerate the semicolon as a command line delimiter (Else and EndIf; also commands run as a single background process and chained with the '+' character). Furthermore, certain commands and directives (like dot-&-space) will not tolerate preceding and/or following tab characters. These quirks are simple enough that, even as bugs, they should not be a problem; but any reader developing scripts under v1.2 who would add comments and/or whitespace characters (other than the space character itself) to script command lines is urged to test for side effects or, alternatively, to put them on separate lines.

• Commands may be indented with the space character.

Finally: readers using word processors to develop scripts should check that file output does not, for example, strip dot directives.

So: where were we? Oh yes...

Passing the current window argument to an input stream parameter has an undesirable side-effect: the current window is no longer available to us as a command line interface. Running the statement (as a background process) won't work either (nor, when you think about it, would we want it to). Running will work, however, if we take the input stream from a console window:

```
1.RAM DISK:> Run Join NewCon:5/80/629/45/my_window as foo
```

(You may prefer to position or size the window differently.)

While this method works, its length makes it hardly more useful than the statement it replaced. The solution, of course, is to execute the statement from within a script.

```
1 . GetText
2 .key AS/a
3 Join "NewCon:5/80/629/45/Enter text (up to 255
  characters between CRs); [CTRL-\] to close." as "<AS>"
```

Line 2 features a parameter declaration or 'template' for the command line argument that is substituted into line 3 - in this case, Join's AS parameter. As with normal AmigaDOS commands, the "/a" suffix indicates that we want this script to fail if it is executed without an argument. Note also that the AS parameter in line 3 is in double quotes, as the user may enter a filename bearing spaces or other special characters.

Statement 3 also writes a prompt to the NewCon: window title bar. Note that the use of space characters in this prompt requires that the entire window specification be enclosed in double quotes.

We can now Run our script by entering:

```
1.RAM DISK:> Run Execute GetText foo
```

Our shell window remains 'live' for other command line processing. Now, we can also create simple text files with Echo:

```
1.RAM DISK:> Echo > foo "There's no foo like an old foo."
```

and we don't have to use the CTRL-\ key combination to send end-of-file. However, Echo's orientation to strings rather than files imposes some limits on the kind of I/O it supports. Indeed, it may seem as though Echo cannot express redirection of input. Entering:

```
1.RAM DISK:> Echo < foo ?
```

results in the response:

```
: Invalid argument to Echo
```

The problem is that the string "There's no foo like an old foo" loses its double quotes inside the file. What we need here is:

```
1.RAM DISK:> Echo > bar "*"There's new bar like no
   old foo.*""
```

by which the asterisked double quotes are written into *bar*, maintaining the integrity of the string. But surely this is a cumbersome way to create a text file. Indeed, one of the nice things about passing text to commands that orient to files for their input streams is the resilience of such commands to metacharacters like newlines and unclosed double-quotes - character sequences that Echo has a hard time understanding, but that are likely to occur (if for no other reason than typing error) when input comes from the keyboard.

Note, by the way, that we can also use pipes as 'throwaway' containers for text. For example:

```
1.RAM DISK:> Echo > Pipe:foo "Echo *"There's new foo
   like no old foo - nor, for that matter, like new
   bar - and ***"Hello, world!***", while we're at it*""
```

Furthermore, we can execute such 'piped' scripts, just as we would the 'filed' variety. Remember, however, that an Amiga Pipe: is a flip-flop kind of device; once 'filled', it can't be filled again until it is 'emptied'; once emptied, it can't be emptied again until it is filled.

2a. In the matter of the '?' argument

Experience tells us that the shell interprets command lines more or less according to the following routine:

1) get the command line;
2) find the file named by the first string in the command line and check that it is executable;
3) load the command from the specified or implied directory;
4) if the '?' argument is present:
   - send the command's template to the output stream given with the command (If no output stream is specified, then the current window is the default);
   - append to the argument(s) given with the command the input stream given with the command (if no input stream is specified, then the current window is the default), and suspend execution until the input stream closes;
   - when the input stream has closed, execute the command.

Many of the scripts here use the '?' argument to force input redirection - sometimes from a window, at other times from the file system. Some exploit the suspension of execution that '?' forces when the console is the input stream. Some specifically exploit the fact that the suspended command (or script) is nonetheless loaded into address space before being suspended. A few combine input redirection with display of a 'user-friendly' or 'descriptive' secondary prompt. And one exploits the redirection of the command template to the file system.

Various commentators have interpreted the integration of the '?' argument into AmigaDOS as a poorly-implemented system of on-line help. Such an interpretation would seem to overlook a substantial portion of its functionality.

2b. More on creating text files with Join and Echo

GetText is limited in that a maximum of 255 characters can be entered in one piece without pressing RETURN. This is not bad, but let's overcome this limitation and demonstrate some other programming techniques in the process.) One way is to take input from more than one window; something like:

```
1 . MoreText
2 .key AS/a
3 Join "NewCon:5/11/629/45/Enter text (up to 255
   characters between CRs); [CTRL-\] to close." "NewCon:
   5/45/629/45/Enter more text (up to 255 characters
   between CRs); [CTRL-\] to close." as "<AS>"
```

But it's hard to see much of a future in this. What we need is a script that accepts input, indefinitely, until the user says to stop. The following script, *Texter*, does just that.

```
1 . Texter
2 .key TO/a,SKIP/s,FILE.0/k,FILE.1/k
3 .bra {
4 .ket }
5 {SKIP}
6 Run > Nil: Echo > Nil: < "Con:5/59/629/25/If no more
   text for {TO}, [RETURN] in this window." ?+
7 Echo > :t/texter.d
8 Lab
9 If exists :t/texter.d
10   Join {FILE.0$Nil:} as "{TO}"
11   Delete > Nil: :t/texter.?
12 Else
13   Join {FILE.0} "NewCon:5/80/629/45/Enter text (up to
      255 characters between CRs); [CTRL]-\ to close." as
      {FILE.1$:t/texter.1}
14   Execute Texter "{TO}" skip file.0
      {FILE.1$:t/texter.1} file.1 {FILE.0$:t/texter.0}
15 EndIf
```

Texter's strategy is two-fold. First, *Texter* uses recursion to create an indefinite sequence of console windows into which text may be entered; as each section of text is closed, it is added to the file of previous sections to make a new aggregate file, and a new window is opened. Second, *Texter* sets going and then suspends execution of a background process which, on re-activation by the user, breaks *Texter*'s recursive loop.

As you might expect, the background process is set up first, in lines 6 and 7. Assuming for the moment that the user has not yet replied to the backgound process's window (back to that in a moment), foreground process control passes to line 13.

Now, on *Texter*'s initial execution, no argument is passed to FILE.0, such that it does not figure in line 13 - it simply drops out. Instead, the user sees the (now familiar) console window, the input to which must be terminated with CTRL-\. In contrast, when (on initial execution) no argument is passed to FILE.1, it also drops out, but with the difference that the temporary file:

```
:t/texter.1
```

is inserted in its place (note the default parameter flag, '$'), and it is to this file that user input is written.

Texter now recurses (line 14), passing to itself whatever the user had given as a TO parameter. Also passed are parameters generated by line 14 itself, including SKIP. *Texter* uses the occurrence of SKIP as a way of detecting whether or not it has been executed from the command line or via recursion; i.e., as a way of ensuring that the background process runs only once, no matter how many times *Texter* recurses.

Returning to line 14: as this is still the initial execution of *Texter*, the arguments accompanying the keywords FILE.0 and FILE.1 consist of the default parameters:

```
:t/texter.1
:t/texter.0
```

respectively; not until the next execution will the arguments for FILE.0 and FILE.1 figure in line 14. Note, however, the association of the keyword "file.0" with the argument

{FILE.1$:t/texter.1} and the association of the "file.1" keyword with the argument {FILE.0$:t/texter.0} - the association of keywords with temporary files is reversed each time *Texter* recurses. It is precisely because of this alternation of keyword association that *Texter* is able to (in line 13) indefinitely concatenate the user's input into one or the other of the files.

Sounds complicated? Let's try that again:

On every odd iteration of *Texter* (the first, third, fifth, etc.), line 13 resolves to:

```
13 Join :t/texter.1 "NewCon:...to close" as :t/texter.0
```

This gets the current contents of :t/texter.1 and some text from the window, and puts the concatenated result in :t/texter.0. In like manner, even iteration's of *Texter* resolve line 13 to:

```
13 Join :t/texter.0 "NewCon:...to close" as :t/texter.1
```

and :t/texter.1 is updated. While the assignment of default arguments to the FILE.0 and FILE.1 parameters is fixed on *Texter*'s initial execution, the passage of arguments to FILE.0 and FILE.1 is thereafter managed dynamically, in consequence of the alternation of association that is performed by line 14.

This process of alternatingly updating *Texter*'s temporary files continues indefinitely - indeed, there is no way out of the loop within the process from which *Texter* was called. Instead, Texter checks (line 9) for the existence of a dummy file:

```
:t/texter.d
```

That may be created by some other concurrent process - specifically, the background process set going by *Texter* itself.

You will remember that Statement 6 remains in suspense until the user makes Echo's window active and types RETURN. (As no text is entered, no use is made of the enhanced console handler.) Execution of the background process resumes, creating (in line 7) the marker file. Control within the foreground process can now pass to line 10, where the most recently updated of *Texter*'s temporary files is copied to TO, the user-specified destination file. Texter cleans up (line 11) and terminates.

Texter represents a considerable increase in complexity over *GetText*, and some of that complexity resides in its details. For example, three out of Texter's four parameters - SKIP, FILE.0 and FILE.1 - are for its own use. Note that SKIP is a 'switch': if "skip" appears as a command line argument, then the keyword occurrence "{SKIP}" in line 5 is 'filled in'; if not, then the keyword occurrence - line 5 itself - is simply left blank.

Statements 3 and 4 define alternate bracket characters. As noted above, the shell uses the characters '<' and '>' to invoke input/output redirection. The same characters are used to distinguish occurrences of keywords in script lines. Without alternate bracket character definition, the shell will not tolerate input redirection (though it will tolerate output redirection). Statement 6 features input redirection; ergo, lines 3 and 4.

Many of the scripts in this article provide for redirection of console output to the NIL: device; for example, *Texter* line 6 provides for output redirection from both Run and Echo. In these and in other cases, console display of secondary prompts or error messages serves no useful purpose in managing user interaction with the script as a whole. On the other hand, line 10's use of NIL: is purely defensive; as it is conceivable that the user could reply to statment 6 fast enough to create :t/texter.d before *Texter* had had a chance to recurse, a default input argument must be provided to take the place of FILE.0.

If *Texter* may claim some elegance in its solution to the problem of indefinite text concatenation, its method of ending recursion is a kludge, without question. (In fact, we shall be able to say this about many of the scripts featured herein.) Furthermore, we might prefer to find some way around the requirement that the user send end-of-file with each section of text; the CTRL-\ key combination is both awkward and non-intuitive. While the use of end-of-file ensures the non-traumatic processing of control sequences and permits both integral newlines and same-line text continuation across section boundaries, we might be willing to trade off some of that functionality for the ease of use that accompanies closing off each section of text with a simple RETURN. Finally, the kind of use made by *Texter* of temporary files is such that we might prefer to replace them with pipes.

Accordingly, *Liner*.

```
1 . Liner
2 .key TO/a,SKIP/s,FILE.0/k,FILE.1/k
3 .bra {
4 .ket }
5 FailAt 21
6 {SKIP}
7 Run > Nil: Echo > Pipe:liner.d < "Con:5/69/629/25/If no
   more text for {TO}, [return] in this window." ?+
8 Join Pipe:liner.d as Nil:
9 Lab
10 Join > Nil: Nil: as Pipe:liner.d
11 If not fail
12   Join {FILE.0$:Nil:} as "{TO}"
13   Join Pipe:liner.d as Nil:
14 Else
15   Execute GetArgs Pipe:liner.n prompt "Enter text for
      {TO} (15 strings 255 characters maximum.)"
16   Join {FILE.0} Pipe:liner.n as {FILE.1$Pipe:liner.1}
17   Execute Liner "{TO}" skip file.0
      {FILE.1$Pipe:liner.1} file.1 {FILE.0$Pipe:liner.0}
18 EndIf
```

Liner is built on the same plan as *Texter*, with a few small but important differences. As the listing shows, temporary files are gone; instead, pipes serve both as text buffers and as markers for the background process. The former is pretty self-

explanatory; the latter introduces some new twists on shell programming.

In line 7, Echo opens a pipe and remains in suspense until the user RETURN's in Echo's window; thereupon (line 8) the pipe is emptied. As noted above, it is a property of pipes that, once a pipe is opened, no other process can fill ituntil it has been emptied. Statement 10 thus fails until the user responds, and *Liner* can detect (line 11) the state of the background process.

But there is another, more fundamental difference. Where *Texter* had taken 'new' user input to Join from a console window, *Liner* takes that input from the pipe:

```
Pipe:liner.n
```

But from where does the pipe get its input?

In line 13, *Liner* calls another script - *GetArgs* - to get user input. (Note that *Liner* also supplies *GetArgs* with the name of its pipe, and a prompt string.) Now, *GetArgs* is executed in the same process as *Liner*: once *Liner* calls *GetArgs*, *Liner* doesn't resume until *GetArgs* has terminated - clearly, *GetArgs* is where the action is.

GetArgs consists of the following rather cryptic statements:

```
1 . GetArgs
2 .key TO/a,PROMPT/k
3 .bra {
4 .ket }
5 FailAt 21
6 Execute > Nil: < "NewCon:5/94/629/25/{PROMPT}" PutArgs
  "{TO}" ?
7 If fail
8   Execute GetArgs "{TO} "Entry not accepted. {PROMPT}"
9 EndIf
```

But *GetArgs* is not the end of the line, either; *GetArgs* calls (line 6) yet another script, *PutArgs*. In fact, even though it is *GetArgs* that opens a window for input, everything entered in the window is passed to *PutArgs* as an argument. Previous scripts have used input redirection to AmigaDOS commands; *GetArgs* uses redirection to a script. Note also how line 6 integrates the arguments passed to *GetArgs*: *Liner*'s "Pipe:liner.n" (*Liner*, line 15) is simply passed on to *PutArgs* as an argument, while *Liner*'s prompt string figures in *GetArgs*' NewCon: title bar.

Why *PutArgs* must be executed in this admittedly convoluted manner will become clearer as we turn to *PutArgs* itself.

```
1 . PutArgs
2 .key _0/a,_1,_2,_3,_4,_5,_6,_7,_8,_9,_A,_B,_C,_D,_E,_F
3 Echo > "<_0>" "<_1> <_2> <_3> <_4> <_5> <_6> <_7> <_8>
  <_9> <_A> <_B> <_C> <_D> <_E> <_F>"
```

When *GetArgs* calls *PutArgs*, *Liner*'s "Pipe:liner.n" becomes the first of any arguments passed to *PutArgs*; i.e., it is associated with the parameter _0, *PutArgs*'s destination file. *PutArgs* writes up to fifteen strings to _0; the others come from *GetArgs*' window. (Why fifteen? Because it seemed like a good idea at the time. Note that if the user enters a string that is identical to any of the keywords in *PutArgs*' declaration, then that string will be assigned to that keyword, and the order of the strings will be corrupted. For this problem there is no solution; all we can do is use relatively obscure keywords such as "_0", which are assumed to be unlikely to occur.)

Now, nothing prevents us from executing *PutArgs* from the command line:

```
1.RAM DISK:> Execute PutArgs foo string_1 ... string_15
```

though Echo, alone, will accomplish the much the same, as seen above. However, whereas Echo requires user-supply of enclosing double quotes, *PutArgs* permits solicitation of user input as a series of separate strings. Furthermore, *GetArgs* provides for recursion (*GetArgs*, lines 5, 7 and 8) should *PutArgs* fail due to faulty user input (metacharacters, too many strings, etc.) In such cases, *GetArgs* tries to assist the user by passing to itself an explanatory prompt (*GetArgs*, line 8); on recursion, this explanation is inserted into *GetArgs*' PROMPT argument.

(Why was no FailAt statement included in *Texter*, which also solicits input via Echo? Because the 'failure level' associated with the Echo command is not so high as to prevent execution of subsequent statements. It may appear as though faulty input to *PutArgs* brings down *GetArgs* as an indirect consequence of the failure of *PutArgs*' Echo, but it is GetArgs' own Execute that rejects the bad input and prevents Echo from even knowing that any input had been received.)

Back now to *Liner*, which in line 13 calls *GetArgs*, passing it the pipename to which input to *GetArgs* is to be directed (Pipe:liner.n), and a prompt string. *GetArgs* simply passes the pipename to *PutArgs* as it sets up the NewCon: window from which the rest of *PutArgs*' input is to come. (*GetArgs*, line 6). *PutArgs* accepts the input, creates the file, and ends, returning control to *GetArgs*. As *GetArgs* has finished its work, control returns to *Liner*. The contents of the now-full pipe are emptied into *Liner*'s current FILE.0 pipe, (Pipe:liner.0 or Pipe:liner.1, depending on which recursion called *GetArgs*. By this doubly-indirect strategy, *Liner* manages line-at-a-time text entry; as all text is handled as string input, end-of-file (CTRL-\) isn't needed.

3. A digression on temporary-file management

Texter and *Liner* depend heavily upon temporary files and pipes. In some cases, these resources are used to 'emulate' flags and variables. Perhaps it seems as though this misses the point of what computing resources are good for in the first place. On the other hand, we might as well find out what the tools can do, even if by sometimes devious means.

A general problem with temporary files and pipes is their vulnerability to corruption in a multi-programming environment.

Consider, for example, the Execute command itself: clearly it would do no good were the temporary, parameter-substitution-resolved version of some script to be clobbered by re-use of the same name as a result of, for example, concurrent execution of the same script. The same may be said of files/pipes used by scripts, regardless of their purpose and, in the end, the overall practicability of scripts that orient to temporary files/pipes in a multi-tasking environment is founded on the possibility of assuring filename uniqueness in some principled manner. Unless this goal can be realized, we will be forced to execute scripts one at a time, à la MS-DOS. Surely we can do better than that. Well, yes, we can. Unfortunately, the details are too lengthy for inclusion here, and will have to await another article.

4. Managing variation in startup conditions

While text entry utilities like *Texter* and *Liner* can be put to use in a variety of contexts, their exploitation of concurrency leads to performance degradation if each member of their command-set must be loaded from floppy disk. If we Resident their command-set at boot time, performance is acceptable. But we might not want to Resident the command-set every time; better that we save the original Startup-sequence as, say, "Startup-sequence.old" and make a new one incorporating the calls to Resident. If we want to go back to our old Startup-sequence, we invent a name for the current Startup-sequence (call it "DoResident-sequence"), and reverse the names:

```
1.RAM DISK:> Rename S:Startup-sequence as
   S:DoResident-sequence
1.RAM DISK:> Rename S:Startup-sequence.old as
   S:Startup-sequence
```

As a more general solution, we could dedicate a directory in :s - call it "seqs" - to storing alternate startup-sequences. We could keep a master of each script in :s/seqs and (to change to a different startup sequence) copy from that to :s/Startup-sequence. This keeps the non-current sequences hidden from view, and less likely to be accidentally deleted or executed.

Unfortunately, this arrangement means that our boot disk contains two copies of the boot-file - a master copy, and the current working copy. Furthermore, it is simply inefficient that we should rewrite the information; what we need is a way to Rename the master as the working copy without losing track of its original (master) name. We could put that name in a file, but managing so many directories and files becomes a source of confusion in itself. What we need is a utility that both remembers the original filename and performs Startup-sequence subsitution. The next script, *ChangeSeq*, does exactly that.

4a. *ChangeSeq* - a Startup-sequence management utility

Given:s/Startup-sequence, as factory-supplied, suppose that there exists an alternate startup-sequence,

```
:s/seqs/DoResident-sequence
```

as already described. Suppose, also, that there exists a file,

```
:s/seqs/current
```

the contents of which are none other than the string:

```
":s/seqs/Startup-sequence.old"
```

That is, suppose that we have written the master pathname given to the factory-supplied startup-sequence to the file :s/seqs/current. (I have adopted the convention that scripts bearing the suffix "-sequence" are scripts written for the purpose of implementing startup conditions.)

In pseudo-code, *ChangeSeq* works as follows:

- check that DoResident-sequence is present in the appropriate subdirectory (:s/seqs);
- retrieve the pathname of the file currently being used as :s/Startup-sequence from :s/seqs/current, and move the file back to the :s/seqs subdirectory (i.e., Rename it as :s/seqs/Startup-sequence.old);
- rename DoResident-sequence as :s/Startup-sequence;
- echo DoResident-sequence's master pathname to :s/seqs/current, so it may be retrieved the next time we run *ChangeSeq*.

Translating all that into AmigaDOS commands, we get:

```
1 . ChangeSeq
2 .key FROM/a
3 .bra {
4 .key }
5 If exists ":s/seqs/{FROM}"
6   Rename > Nil: < :s/seqs/current from
      :s/Startup-sequence ?
7   Rename ":s/seqs/{FROM}" as :s/Startup-sequence
8   Echo > :s/seqs/current "*":s/seqs/{FROM}*""
```

ChangeSeq closely follows the model of its pseudo-code. In line 6, the inclusion of the keyword FROM serves only to articulate how command line and redirected input are associated with Rename's arguments; it is not strictly necessary.

As made clear by the pseudo-code, line 8 stores the 'real' name of the current startup-sequence in :s/seqs/current. Again, the use of special characters to write quotation marks into the file itself protects against the use of spaces in the filename.

Now, *ChangeSeq* is not so bad as far as it goes, but :s/Startup-sequence is not the only boot-time-relevant file that benefits from this kind of management. For example, there are times when I don't want to display the mouse pointer; yes, a program could be written to change it to something else, but we can also write a *ChangeConf* that does much the same work.

The same could be said for any 'hardwired' file (:s/Shell-startup, for example), or for 'softwired' files for which similar

file management techniques are appropriate (:devs/Mountlist, for example). But it seems somehow wrong to write several scripts to do the work that should be able to be done by one; accordingly, we will make some modifications to *ChangeSeq*.

4b. *ChangeBoot* - a boot-file management utility

We could write all sorts of variation into *ChangeBoot* but, for purposes of illustration, we will assume, first, that there exist the following files:

```
:devs/confs/current
:s/seqs/current
:s/starts/current
```

and second, that each of these files contains the 'real' name of the boot-file currently in use; and third, that the directories :devs/confs, :s/seqs and :s/starts contain non-current alternate boot-files.

A listing of *ChangeBoot* follows:

```
1 . ChangeBoot
2 .key "FROM/a, devs/System=configuration=devs/confs/s,
      s/Startup=sequence=s/seqs/s,
      s/Shell=startup=s/starts/s"
3 .bra {
4 .ket }
5 If exists :{devs/confs}{s/seqs}{s/starts}/{FROM}
6   Rename > Nil: <
      :{devs/confs}{s/seqs}{s/starts}/current from
      :{devs/System}{s/Startup}{s/Shell}-
      {configuration}{sequence}{startup} ?
7   Rename :{devs/confs}{s/seqs}{s/starts}/{FROM} as
      :{devs/System}{s/Startup}{s/Shell}-
      {configuration}{sequence}{startup}
8   Echo > :{devs/confs}{s/seqs}{s/starts}/current
      "*":{devs/confs}{s/seqs}{s/starts}/{FROM}*""
9 Else
10   Echo ":{devs/confs}{s/seqs}{s/starts}/{FROM}
      not found"
11 EndIf
```

ChangeBoot takes two arguments, in any order. The first identifies which type of boot-file is being managed. The second is the name of the alternate boot-file that is to be Renamed as current. *ChangeBoot* looks for this file in whatever subdirectory has been implicated by the first argument. If *ChangeBoot* can't find the file, it displays a message. Note that *ChangeBoot* performs its operations in the current volume only.

ChangeBoot looks complicated because the file/directory references of its command statements change according to which of the three boot-files is being managed. By declaring 'keyword equivalents' (line 2), *ChangeBoot* accommodates variation in how arguments are actually used in individual statements. (Restricted keyword length forces us to adopt the devious tactic of representing filenames like "devs/System-configuration" as "devs/System"-"configuration" - though this particular scheme is only useful for boot-files that include hyphens.) As *ChangeBoot* only manages one boot file at a time, most of the bracketed parameters drop out, leaving behind only parameters relevant to the current boot-file. For example, substituting the arguments given in Section 2a, we see that these statements resolve to:

```
5 If exists ":s/seqs/DoResident-sequence"
6 Rename > Nil: < :s/seqs/current from
      :s/Startup-sequence ?
7 Rename ":s/seqs/DoResident-sequence" as
      :s/Startup-sequence
8 Echo > :s/seqs/current "*"
      :s/seqs/DoResident-sequence*""
```

This is identical to our first version.

New to *ChangeBoot* is the enclosure of all parameter declarations in double-quotation marks. The combined presence of equals signs ('=') and multiple parameters requires the amalgamation of the entire specification into a single string. As we shall see, this string may encompass a substantial amount of non-keyword text, as well.

ChangeBoot gives us a tool with which to manage variation in startup conditions; however, *ChangeBoot* requires that the system be running before any changes can be made. This is frequently inconvenient. What we need is a startup sequence (does this come as a surprise!?) that allows us to choose between default startup conditions and conditions identified from the command line. It would also help if we could run this startup sequence in a 'defaults only" mode - without having to sanction each default parameter individually. The remainder of this article will be concerned with working out a shell script with which to manage conditions implemented during the startup sequence.

5. A multi-purpose startup sequence

The following startup sequence makes relatively few assumptions about the physical configuration of the system and the manner in which system-relevant files are distributed across drives or volumes. Furthermore, what assumptions it does make may be altered, during startup, by execution of the sequence in an 'interactive' mode. Finally, its design is modular, and may easily be modified or expanded to index different sorts of startup conditions as the consequences of its default-mode execution.

It may be worth noting at the outset that the 'startup sequence' being described here is not a single script, but a linked series of scripts, beginning with :s/Startup-sequence on the boot disk, and ending (eight scripts later) with :s/System-sequence. Furthermore, the beginning and ending scripts are not even stored on the same volume. As will become clearer, it is in virtue of this linked mode of construction that the startup sequence as a whole is able to bridge volumes without knowing the name of

the volume to which it bridges, and to combine default and interactive execution modes.

5a. Assumptions and conventions

Executed in its default mode, this startup sequence makes certain assumptions. These assumptions include:

- that startup-relevant files are distributed across four disks, the volume names of which are: Startup, Commands, Workbench and IFFfonts
- that volume Startup includes directories devs, l, libs and s
- that volume Commands includes directories c and s
- that the :s directory on volume Startup and the :c directory on volume Commands contain the commands necessary for the execution of their respective script files; and,
- that the :s directories on volumes Startup and Commands contain the script files described below.

It is also assumed (though the script files described below do not require it) that volume Workbench includes system-relevant, icon-executable files such as :System/Format and :Utilities/Calculator. Note, again, that these assumptions figure only in how these scripts execute in their respective default modes. That the user could be freed from being locked in to a particular set of startup conditions was a major design goal in the development of this sequence. At the same time, it is no accident that the default execution of this sequence has the particular consequences it does.

Beyond serving to demonstrate a set of programming techniques, the distribution of directories and files across disks implied by such reflects an interest in how system-relevant files might be most effectively consolidated within a memory-poor Amiga configuration. Why consolidate system-relevant files in this particular manner? That discussion is beyond the scope of this article; however, any reader who has had to interrogate half a dozen disks for the simple purpose of finding a common AmigaDOS command will have some idea of the concerns at issue here. What is intended is that readers conceive a notion of how this sequence may be extended to comprehend the particular startup options that are appropriate to their situation.

5b. Principle of operation

We may think of the startup sequence as a compound script, consisting of:

```
Startup:s/Startup-sequence     Startup:s/Shell-sequence
Startup:s/Default-sequence     Startup:s/Workbench-sequence
Startup:s/Commands-sequence    Commands:s/Startup-sequence
Commands:s/Fonts-sequence      Commands:s/System-sequence
```

Startup:s/Shell-sequence executes Startup:s/Default-sequence so as to leave the input stream open, as we will see below, using:

```
Execute Startup:s/Default-sequence ?
```

As will become clear, Startup:s/Default-sequence is waiting for one of two sorts of responses:

1. the character 'd' or the string "default", plus RETURN
2. anything else (except special characters), plus RETURN

Upon some user response, the input stream closes, and the remainder of Startup:s/Default-sequence is executed, including a call to Startup:s/Workbench-sequence. It is the central feature of this startup sequence as a whole that in what mode Startup:s/Workbench-sequence is executed - with or without the '?' argument - depends on the user's earlier response to Startup:s/Default-sequence's prompt.

You say this sounds confusing? Let's try again:

The last statement in Startup:s/Default-sequence consists of:

```
Execute Startup:s/Workbench-sequence <DEFAULT$?>
```

Now, whether Startup:s/Workbench-sequence is executed with the '?' argument depends on whether the user enters the character 'd' (or the string "default") in response to Startup:s/Default-sequence's prompt. If 'd' (or "default") is entered, then parameter substitution resolves Startup:s/Default-sequence's last statement as follows:

```
Execute Startup:s/Workbench-sequence DEFAULT
```

and Startup:s/Workbench-sequence is executed without the '?' argument - the input stream remains closed to the user. If, on the other hand, no suitable string was entered at the initial prompt, then the above statement, turns into:

```
Execute Startup:s/Workbench-sequence ?
```

and Startup:s/Workbench-sequence seeks additional input from the console. It is in consequence of this modification to Startup:s/Workbench-sequence's mode of execution that the user has the opportunity - at exactly this juncture - to perform parametric substitution from the command line.

It may be worth noting that Startup:s/Workbench-sequence makes no use of the "default" string, other than to pass it to Startup:s/Commands-sequence, which passes it to a subsequent script, and so forth. Its sole purpose (if entered) is to 'switch off' the '?' argument and thereby, for subsequent scripts, invoke default execution. So what about managing the consequences of the startup-sequence - the reason why we made it interactive in the first place?

In addition to the "default" string input described above, each script (subsequent to Startup:s/Default-sequence) takes a 'substantive' argument; i.e., an argument which, if supplied, alters the execution of the startup sequence in some substantive manner. As noted above, this could be (but is not limited to) assigning different logical directories, adding different numbers of buffers, and mounting different devices. (How these

possibilities are implemented as arguments and statements depends, of course, on the startup conditions to be realized.)

5c. Detailed analysis

The first script in the startup sequence consists, of course, of Startup:s/Startup-sequence.

```
1 ; Startup:s/Startup-sequence
2 Run > Nil: SetAlert      ; all typical v1.3 stuff that
3 FastMemFirst             ; either should happen right
4 BindDrivers              ; away of might as well as it
5 AddBuffers Startup: 5    ; draws on directories included
6 FF > Nil: -0             ; on volume Startup. Note that
7 SetMap usa1              ; commands included on Startup:
8 SetClock > Nil: load     ; are retrieved from the root
9 Resident CLI L:Shell-Seg system pure
                           ; directory (not from :c)
10 Resident Resident pure
11 Mount NewCon:
12 NewShell NewCon:0/200/640/200/MyShell from
     Startup:s/Shell-sequence
13 EndCLI > Nil:
```

In line 10, Resident is made resident; other commands will follow, below. However, to exploit resident commands, we must activate the enhanced (v1.3) shell; i.e., once we have run the commands that are (a) never run more than once, and (b) invariantly associated with startup, we want to get out of the CLI and into the enhanced shell - even though Startup:s/Shell-sequence cannot be execute (line 12) so as to leave the input stream open to the user.

```
1 ; Startup:s/Shell-sequence
2 Resident Startup:Execute
3 Resident Startup:CD
4 CD Ram:
5 Execute Startup:s/Default-sequence ?
```

Startup:s/Shell-sequence exploits command residence to execute what few remaining basic startup conditions are implemented prior to the execution of Startup:s/Default-sequence; i.e., prior to user interaction. One inevitable consequence is the creation, by Execute, of temporary files; the immediately compelling reason to relocate the process to RAM: (line 4) is to avoid setting :t in volume Startup.

```
1 . Startup:s/Default-sequence
2 .key "D=DEFAULT/s"
3 Execute Startup:s/Workbench-sequence <DEFAULT$?>
```

As has already been made clear, Startup:s/Default-sequence's reason for being is to structure the user's choice between running (subsequent) scripts in default or interactive mode. Having said that, it bears noting that we don't really need a separate script to give the user this choice; the presence of D=DEFAULT as a parameter in each subsequent script in the startup sequence implies that choice with each prompt. On the other hand, articulating this choice in a separate script also allows for the display of a specific descriptive prompt (see section 5d, below.)

```
1 . Startup:s/Workbench-sequence
2 .key "D=DEFAULT/s,L=LOADWB/s"
3 <LOADWB>
     ; As LoadWb is on boot disk, pathname not required
4 Execute Startup:s/Commands-sequence <DEFAULT$?>
```

As described above, user entry of "default" as a command line argument to Startup:s/Default-sequence causes "default" to be passed through Startup:s/Workbench-sequence to its successor. Executed as such, Startup:s/Workbench-sequence denies the user the opportunity to choose to load the Workbench screen environment (at least until the startup sequence has finished). Executed interactively, the user is given a choice: '1' or "LoadWb" to load the Workbench, RETURN to pass. For many memory-poor Amiga users, to load or not to load Workbench is very much a question. As one who appreciates and recommends the Workbench screen environment, I wish this was not the case; but if the question is going to be asked at all, it saves disk swapping to ask it now, early in the game.

The next script is somewhat more complex:

```
1 . Startup:s/Commands-sequence
2 .key ",D=DEFAULT/s"
3 "<$Commands:>c/AddBuffers" > Nil: "<$Commands:>" 5
4 Resident "<$Commands:>c/Assign" pure
5 Assign C: "<$Commands:>c"
6 Resident C:Echo pure
        ;
        ; your favourite commands
        ;
12 Resident C:MakeDir pure
13 MakeDir Ram:env
14 ; Ram:t automatically created by Execute
     Startup:s/Default-sequence
15 Assign Env: Ram:env
16 Assign T: Ram:t
17 Resident C:Path pure
18 Path Ram: C: add
        ;
        ; more of your favourite commands
        ;
24 Execute "<$Commands:>s/Startup-sequence"  <DEFAULT$?>
25 Resident Path Remove   ;
26 Resident Assign Remove
     ; not needed after startup conditions realized
```

There are several things new about this script, though - for our purposes - the main interest is the introduction of a somewhat unusual parameter: a parameter declared only in virtue of the comma by which it is distinguished from DEFAULT. That's right: the keyword, or placeholder for all substantive modifications to the current script is undefined with respect to pattern-matching; its sole pattern-matching property consists of the

two strings that it will not match: 'd' and "default." Close examination of the script will reveal that this 'null' parameter occurs five times in the body (lines 3-25) of the script.

Startup:s/Commands-sequence is stored on volume Startup. As suggested earlier on, it is a feature of the manner in which scripts are executed that they are brought into address space before parameter substitution is resolved.

An attractive consequence of this is that, having loaded Startup:s/Commands-sequence, volume Startup may be removed from the drive, and volume Commands (or whatever volume contains the directory to which *c:* will be assigned) inserted. Constructing Startup:s/Commands-sequence such that references to volume Commands (or other disk) do not need to be 'filled in' until volume Commands is actually required yields a major payoff in avoiding redundant disk swaps.

The purpose of Startup:s/Commands is to make resident some of the more commonly used commands, plus those that are necessary for the efficient progress of the remainder of the startup sequence. As certain commands are brought into address space, a number of subsidiary startup conditions are realized; for example, as of line 18, the effective search path for non-resident commands is: the current working directory (currently RAM:), then RAM:, then Commands:c, then C: (currently Commands:c).

The apparent redundancy of this path is misleading. As Sys:System and Sys:Utilities no longer reside on the same disk as c:, it is a matter of some importance that RAM: and Commands:c be added to the path at this time. Note, incidentally, that Path cannot be Run as a background process (which makes sense when you thing about it.)

Note that line 24 assumes that there exists an :s/Startup-sequence on the root level of the volume to which c: is assigned. For what it's worth, this facilitates the execution of a factory-supplied :s/Startup-sequence on, for example, an application-oriented disk (as long as the disk possesses Startup:s/Commands-sequence's command set).

Supposing, however, that c: is assigned to Commands:c, then the execution of Commands:s/Startup-sequence would be as follows.

```
1 . Commands:s/Startup-sequence
2 .key ",D=DEFAULT/s"
3 FailAt 21
4 Assign S: "<$Commands:>s"
5 Path S: add
6 Run > Nil: Execute DoDate
      ; I/O such that it's OK to run concurrently
7 Execute Fonts-sequence <DEFAULT$?>
```

For the benefit of those users who lack a clock, Commands:s/Startup-sequence introduces a subsidiary script *DoDate* (which is assumed to be in the directory to which s: is assigned; though, if it isn't, the main script will continue anyway). As it is given that the volume on which *DoDate* is stored will be mounted during Commands:s/Startup-sequence's execution, this is a good (i.e., fewer swaps) opportunity to execute a subsidiary script as well.

Again assuming that the user has assigned s: to volume Commands, Commands:s/Fonts-sequence could be written as follows:

```
1 . Commands:s/Fonts-sequence
2 .key ",D=DEFAULT/s"
3 Run > Nil: Assign Fonts: "<$IFFfonts:>fonts"
4 Execute System-sequence <DEFAULT$?>
```

Fonts: is assigned in a background process (line 3) so as not to hold back Commands:s/System-sequence.

```
1 . Commands:s/System-sequence
2 .key ",D=DEFAULT/s"
3 FailAt 21
4 Assign Sys: "<$Workbench:>"
5 Path Sys:System Sys:Utilities Sys: add
6 ;Execute -sequence <DEFAULT$?>     ;more scripts ?
```

Since it is possible that the volume to which the user assigns Sys: lacks one or more of the directories named in line 5, FailAt is once again necessary (line 3). Note that, with line 5, the Path additions in this startup sequence are consistent in content, if not order, with those of the 'official' Amiga Workbench startup sequence.

5d. Improving the descriptive power of secondary prompts

All normal AmigaDOS commands and scripts executed normally with the '?' argument send a template to the console (to indicate to the user the kind of input being sought).

The startup-sequence described above does not conform to this practice in a meaningful way. Supposing that this is a problem, our solution will be to use console control sequences to embed and format informative text within the secondary prompt, and at the same time hide some of these script's complexity from the user.

Here are three examples of startup sequence scripts rewritten to exploit control sequences: First, Startup:s/Default-sequence:

```
1 . Startup:s/Default-sequence
2 .key "ESC[32m:ESC[31m enter
      ESC[1;32mdESC[0m=ESC[1;32m DEFAULTESC[0m for
      default startup sequence |
      ESC[4mreturnESC[0m to
      modifyESC,D=DEFAULT/s,ESC[32mESC[E "
3 Execute Startup:s/Workbench-sequence <DEFAULT$?>
        ;
        ; etc.
        ;
```

The prompt generated by this script looks something like this:

```
: enter d=DEFAULT for default startup sequence |
      return to modify
:
```

where the colons, the single character 'd', and the string "DEFAULT" are rendered in colour 2. The functional portion of the declaration, being rendered in colour 0, does not appear.

Second, Startup:s/Workbench-sequence:

```
1 . Startup:s/Workbench-sequence
2 .key "ESC[32m :ESC[31m enter
      ESC[1;32m1ESC[0m=ESC[1;32m LoadWbESC[0m to load
      Workbench screen environment |
      ESC[4mreturnESC[0m for CLI onlyESC,D=DEFAULT/s,
      L=LOADWB/s, ESC[32mESC[E "
3 <LOADWB>
4 Execute Startup:s/Commands-sequence <DEFAULT$?>
        ;
        ; etc.
        ;
```

where the prompt looks something like this:

```
:enter l=LoadWb to load Workbench screen environment |
      return for CLI only
:
```

Our last example of control sequence integration is Startup:s/Commands-sequence.

```
1 . Startup:s/Commands-sequence
2 .key "ESC,ESC[32m:ESC[31m enter
      ESC[3;32mroot:ESC[0m to assign C: to volume
      ESC[3mroot:ESC[0mc | ESC[4mreturnESC[0m for
      ESC[1;3mCommands:ESC[0mcESC,D=DEFAULT/s,
      ESC[32mESC[E        "
3 "<ESC$Commands:>c/AddBuffers" > Nil:
      "<ESC$Commands:>c" 5
4 Resident "<ESC$Commands:>c/Assign" pure"
        ;
        ;
        ;
24 Execute "<ESC$Commands:>s/Startup-sequence"
      <DEFAULT$?>
        ;
        ;
```

for which the prompt consists of:

```
: enter root: to assign C: to volume root: |
      return for Commands:c
:
```

in which the colons and the initial instance of the string "root" are rendered in colour 2.

In its initial (non-control-sequence) version, this script (and its successors) associated command line arguments other than 'd' and "default" with a null parameter. Above, that null parameter is transformed into a series of non-printing characters: the control sequence itself. That is, the placeholder for substantive command line arguments is the string:

```
ESC
```

After all, who ever said that script keywords had to be printing characters?

5e. *DoDate* - a Run-nable Date utility

DoDate appears in Commands:s/Startup-sequence, above.

```
1 . DoDate
2 .bra {
3 .ket }
4 FailAt 21
5 Date > Nil: < "NewCon:5/90/629/25/DoDate: enter
      date (DD-MMM-YY) and/or time (HH:MM); [return]
      to resume." ?
6 If fail
7   Execute DoDate
8 EndIf
```

DoDate recurses until the user offers satisfactory arguments to the Date command.

5f. Summary

Were this startup sequence not interactive, it would still comprise many statements. Tweaking script statement order to exploit caching and command residence reduces, but does not eliminate, the user's time/energy investment. Users who spend most of their time with a single application will likely feel that an investment of this magnitude just doesn't make sense. The same may be true of users who, whether by need of contiguous memory, or simple habit, reboot on quitting any given application.

On the other hand, users who must manipulate system conditions before setting up their application may find some aspect of this model worth imitating. Not demonstrated but quite conceivable is the extension of the techniques illustrated here to articulate a decision tree of startup conditions.

As programs go, shell programs are high-overhead and limited in scope. Whether a shell program even makes sense as an approach to a particular programming task depends on the pattern of one's computer use and the level of abstraction of the task at hand. Where such considerations point to the viability of shell programming, then shell programs are far more easily implemented than their C or BASIC equivalents. Their use requires a certain understanding of AmigaDOS concepts, and the availability of models. I hope that the models provided in this article will prove helpful to others who wish to explore shell programming.

WordPerfect's new Amiga Workbench

A review of WordPerfect Library

by Jay A. Gross

After almost a year of updating, WordPerfect on the Amiga is still not nearly perfect, not nearly an Amiga program. Now there's WordPerfect Library, a newborn litter of spotted puppies from the WordPerfect kennels.

Basically, WordPerfect Library adds a kit of utilities to the WordPerfect environment so Amiga owners won't have to use anything in their systems that WordPerfect doesn't sell. Library isn't completely Amiga-esque. Its multitasking is very limited, and it uses WordPerfect's same, incredibly sluggish routines for getting imitation Amiga requesters on the screen. To coin a phrase, the utilities have "no respect." They eat CPU power voraciously, making multitasking unpleasant even for their WordPerfect siblings.

Yet, Library has many redeeming qualities. There are two and a half rather nice programs in the batch, and while WordPerfect software products are not perfect, WordPerfect support is.

Library's nicest program, and its most useful, is Calendar, which itself comprises an appointment book, a 'To Do' pad, a Memo pad, and an Alarm program. These sub-programs appear in Amiga-looking, sizeable, movable windows and they can be opened and closed individually; either from a pulldown menu or with a function key (WordPerfect's stock-in-trade). If you have used Calendar on MS-DOS machines, you'll feel right at home; the key commands are identical, as with all of the Library programs.

Just as the kit of windows can be opened from a pulldown menu, they can also be closed that way - all at once, or by clicking the close gadget on each one. Better use the gadgets. The "close all" option from the pulldowns leaves several hundred bytes of memory allocated somewhere that won't be returned to your system. Since Calendar is too much of a CPU hog to leave running in the background, you'll have to close and open it repeatedly, too - chewing up bits of memory each time. The least you can lose seems to be 200 bytes. This does not count the 26K library that all WordPerfect applications share. The system returns memory used by shared libraries after they're freed if it needs the space, and WordPerfect Library's .library apparently behaves properly in that respect.

Calendar is a useful program with no direct competition on the Amiga. It would be really helpful for business types. When the computer is fired up first thing in the morning, the day's Calendar would be displayed, reminding with its Memo window, its To-Do list, and Appointment list, and sounding the alarm at appropriate times. Alarm can run independently of the other programs, and its method of attracting your attention can be the usual beep, or a stirring speech by the Amiga narrator, or any digitized sound. Get it? Any digitized sound. Use your imagination and keep life interesting around the old office.

Calendar offers a facility for calculating how many whats are between two dates. How many Sundays, or how many Wednesdays, for example, between August tenth and October the first? Amiga user group meetings and other important, recurring appointments can be entered batch-wise with a handy menu. Ignore all the old-tech keyboard entry stuff in those menus and just use the mouse to click where you want to put something. It's the Amiga way. The keyboard stuff is a confusing annoyance here.

Calendar permits merging its files, and is supposed to flag conflicting appointments whether in the merged file or normally. In the initial release of the product, this feature doesn't function; but an update and bug fix is already in the works. You can count on WordPerfect for updates - part of the company's stellar user support. Calendar's several windows are easily configured to personal taste and a "save default" feature will make them come up that way henceforth. A word of wisdom: if the Library programs ask you for a disk, feed it the disk. Cancelling the request sends an engraved invitation to the guru under many circumstances.

Like other WordPerfect applications, Calendar has a handy zoom gadget in addition to its front/back buttons. Fully zoomed out, the window occupies the entire screen - even on a high-res WorkBench - and the program puts reminders on a block diagram of the current month to indicate whether there is a To-Do list, an Appointment to keep, or a Memo for that day. It will also display a key word from the day's Memo - "meeting" or "vacation" - if the display is large enough. The display does not adjust correctly to a high-res WorkBench, however, and the system font absolutely must be a vanilla eight-line one. Topaz-11 will make a mess of any WordPerfect

display window, including Calendar. Using the To-Do list, Appointments and Memo windows is troublesome with the Calendar at full zoom, too; since the daughter windows don't pop to the front when menu items such as "add/edit memo" are selected.

Another small problem is that the current time of day should be displayed on the parent task, not on each and every one of the subtasks. Forget a fix ever being done on this one, though, since WordPerfect's aim in life is to make the Amiga look like an IBM, and the IBM version probably does the same stupid thing.

WordPerfect's standard version of the Amiga proportional gadget is a random block with no rhyme or reason for anything it does. The arrows don't always repeat; the slider hops around like an crazed toad, and besides that, no two of the company's proportional gadgets work the same way - if they work at all. Strangely, Calendar's proportional gadget in the Appointments window works just about like Amiga ones are supposed to. It's quite a shock. Has someone at WordPerfect Corporation learned to operate an Amiga?

Calendar is not completely friendly to the Amiga multitasking environment since it soaks up a good bit of the CPU's power for itself if left parked in the machine while other things occur. This is a serious failing of all of the WordPerfect programs, especially WordPerfect itself. For example, although WordPerfect supports up to 32 (re-entrant) document windows at once, the Amiga operates so slowly that it looks like it's crashed after only three documents are running. Library's applications count off the same high overhead, so if you have Calendar, Calculator and two WordPerfect documents operational, you can go out for coffee and Twinkies while the system opens a window.

Notwithstanding the presence of front/back gadgets on all of its windows, Library offers in its pulldown menus (as well as the ubiquitous F-keys) a "switch" option. This brings up a window (none dare call it "Requester") listing all available WordPerfect tasks, from which you can pick one to switch to, using either the mouse or those infernal key commands. Stupidly, WordPerfect's "switch" option will put up a list of only one place to go if that's all there is. Moreover, the "switch" window offers only WordPerfect applications as destinations.

The Amiga comes with a free calculator program that sleep()s peacefully on the WorkBench until you need it. The Amiga Calculator, though, doesn't know hexadecimal, and never heard of Future Value or Straightline Depreciation. Enter WordPerfect with the perfect (?) solution: another of the Library litter, Calculator. Flick between Scientific, Programmer, and Financial calculators at the click of a mousebutton (mousebutton? heresy!).

WordPerfect's Calculator is another pretty pup. It works fine even before the first update, and has enough options and functions to keep Einstein, Kernighan, and Boesky all entertained and happy. A particularly stellar feature is its window of registers which can remain accessible even after Calculator is closed. As far as calculators go, Gizmoz (Digital Creations), from 'way back in the infancy of the Amiga, has a neat approach: three separate calculators, one each for Scientific, Programmer, and Financial, with a storable "tape" that prints to screen or printer. WordPerfect's calculators are better, but they're bound into one program that's too unwieldy to leave wallowing around in the machine all the time.

Library's FileManager is a utility for moving disk files around without coping with the CLI environment. FileManager is well organized and well done. However, it provides no facility for viewing an IFF file (predictable, since MS-DOS has no such graphics file standard). The program is the least DOS-like of the Library applications. Its excellent "Look" option for reading the contents of a file almost makes up for the lack of IFF file viewing. The option permits WordPerfect-style searching, which is a powerful search, indeed.

FileManager's filename match patterns are not Amiga patterns. ARP (Amiga Replacement Project) solves this pattern dilemma by honouring both Amiga ones and the simpler MS-DOS ones as a crutch. WordPerfect Library speakee only asteriskee.

The program also brings something revolutionary to a WordPerfect application: interactive file requesting (not to say "Requester"). Windows of stuff in FileManager are "hot" only after a full screen's worth of filenames is retrieved from the disk. It's a great improvement over WordPerfect, which has non-interactive (read: dumb and slow) filename windows.

The documentation for all the programs, as is usual with WordPerfect products, is voluminous, classy, and outstanding. The package would benefit greatly from a script for installing on a hard drive; although following the directions carefully should not be too difficult - even for novices - and any problems are readily solvable with a call to WordPerfect's toll-free help line. The WordPerfect Amiga help staff has Amigas in front of them, with which to walk callers through any operation if necessary.

Calendar, Calculator, and FileManager are the pick of the Library litter. Three out of five isn't a bad average, and at a retail of $129.95 US, those three, especially Calendar, are probably worth the whole price. You get the others, PEdit and NoteBook, thrown in for what they're worth, which isn't much.

PEdit is a streamlined version of WordPerfect complete with powerful macros (the same as WordPerfect's), fast, fabulous and furious search and replace, and some special programmer-esque tricks, the hex edit one being probably the most useful. Those proportional gadgets mentioned earlier? PEdit's has to be turned on before it bothers you with its presence. WordPerfect's documentation says leaving it off speeds up the program, and it does. Funny, other Amiga editors have no prob-

lem with text speed *with* a proper Amiga proportional gadget running. Without the gadget, though, PEdit operates considerably faster than WordPerfect itself. Just don't open more than one window (it allows 32!). For general text editing, rather than programming, PEdit will be somewhat frustrating, as there is no way to set a line length for word wrap. That's great for C, but not too useful for truly "personal" editing.

PEdit does not support Amiga's AREXX macro language for interprocess communication. It should. TxEd Plus does. So does ASDG's new CygnusEd. Programmers, particularly, will find AREXX useful. Since REXX, of which AREXX is the Amiga implementation, is an IBM darling, maybe WordPerfect can some day be convinced to support its Amiga incarnation.

The shabby condition of PEdit shows again that WordPerfect's recycled MS-DOS crew hasn't figured out a lot of things about the Amiga. For example, multiple select for Bold, Italic, Underline, etc., doesn't indicate anything in the menu bar (a checkmark is usual). The program accepts multiple clicks as though multiple selections were being made, but only executes what was first clicked, and does so even if the pointer is removed from the menu bar with nothing actually selected. For people accustomed to real Amiga programs, this misfeature will be annoying. On the other hand, graduates of the WordPerfect School of Interminable Keyboard Commands will never know the difference.

PEdit eats at least 140 bytes of memory every time it's called up, used a while and then closed down.

Calendar comprises an appointment book, 'To Do' pad, Memo pad, and an Alarm...

The worst of the Library user interfaces is on a potentially very useful program, NoteBook. Martians will have no trouble running it. Amigoids will recoil in horror. WordPerfect fans and masochists will be ecstatic.

NoteBook is a quickie database manager. It keeps track of information, and if you tell it to - carefully, lest the dreaded 'software error' box appear! - it will dial phone numbers. You *do* have a modem attached? The dialing feature is extremely powerful and configurable, permitting any number of variations on a theme. You can, for example, have the program dial the contents of a field, or dial an access number (e.g., a long-distance service) and then the contents of the field, then some more access codes or extensions, or any number of other possibilities. The program does little error checking before the modem is told to dial, and if it's told to do something it can't do, there will be a problem, possibly even a 'software error' box. A little checking on the string sent to the modem would be nice. Experienced modem users will have no problem, but novices will be bald-headed in a week.

The cryptic, terminally unfriendly menus with which this feature is set up could be greatly improved, but don't count on it, since improvement would make the Amiga program different from the MS-DOS ones (better!), a fate worse than debt.

Oh, but if that were NoteBook's only problem. Remember the old days, when you had to memorize a bunch of gibberish command sequences? Those dog days visit us again with Library's NoteBook. The program makes minimalist use of the mouse, and its setup screens are driven by the keyboard in the most un-Amiga manner you could possibly imagine.

Like the other Library utilities, NoteBook supports WordPerfect's print, search, and macro operations where appropriate. Searching for a string within a file listing can be quite handy, and searching Calendar's files can be useful, too. Macros are always handy things to have, and WordPerfect products share a very powerful macro language. In addition, if you can ever get past the horrible user interface, NoteBook will generate WordPerfect compatible merge files. These are not difficult to do in the wordprocessor, but having them done automatically will save time and trouble.

Other caveats. The WordPerfect Print program - the same one for all WordPerfect applications - is set up as a separate task which operates in the background, crashing at the slightest provocation. If you're going to do any printing, make sure you're not doing anything important in the other windows while it's going on, or save your work frequently, just in case. After almost a year of beta-testing (by the *customers*!), WordPerfect's Print program still needs a lot of work. The most crashes will occur in trying to print to a file, especially to RAMdisks or recoverable RAMdisks. If the Print program locks up, you cannot just open another one, as you could with most real Amiga programs. WordPerfect permits only one of its Print programs to operate at once, and when it locks up, you'll have to reboot your Amiga before you can print from WordPerfect.

Summary

On MS-DOS machines - those are the ones with all the keyboard templates taped around the function keys - WordPerfect Library lets single-tasking users flip readily among a number of applications. The background tasks freeze while the active one is in use. On the Amiga, users are free to choose among a variety of applications from any number of manufacturers, and the applications do not have to freeze in the background. Somebody please tell this to WordPerfect. In time, maybe, with continued hassling, WordPerfect's programs might become more Amiga friendly, especially in view of the window/icon/mouse revolution going on in the MS-DOS world. As of now, though, the Amiga is an environment about which WordPerfect has much to learn.

Although WordPerfect Library is probably worth what it costs, all of the utilities could be better and probably will be, given the company's longstanding reputation for improving and supporting their products. Overall, Library is in fact much more of an Amiga product than WordPerfect itself.

ARexx

The REXX Language for the Amiga

ARexx is a multitasking implementation of the REXX language, an elegant high-level language especially suited for macro-processing. Its clean, simple syntax makes REXX easy to learn for novices — and experienced programmers will appreciate advanced features like compound variables, INTERPRET instruction, and source-level debugging.

- Interactive, Interpreted Operation
- Exceptional String-handling Facilities
- Built-In Library with Over 75 Functions
- Built-In Source-Level Debugger
- Compact, Reentrant code — Only 32K

The Emerging Standard

ARexx is the *de facto* standard for inter-program communication. Software that supports the ARexx command interface can be customized, extended, and combined with other ARexx-compatible software in integrated applications. Watch for announcements of ARexx support from a growing list of Amiga software vendors!

Available Now ... Only $49.95

WShell

The Command Shell You've Always Wanted

WShell is the best command shell on the market, with features and convenience of use that go well beyond the CLI and 1.3 AmigaShell. And yet it's highly compatible with the CLI, so you don't have to learn a new command language to use it!

- Line Editing/Command History/Aliases
- Built-In Commands/Resident Commands
- Prompt String/Window Title "Variables"
- Concurrent Piping
- Transparent Support for ARexx Macros
- Script-bit Support for "Execute" Scripts
- Compact, Reentrant Code — Only 10K

ARexx Command Interface

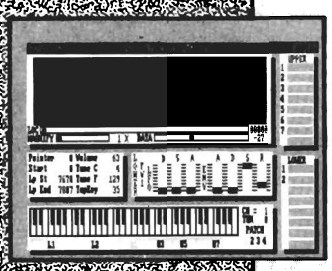
With WShell you can use REXX-language macro programs like executables — even as filter programs in a piping system. Use the source-level debugging to single-step through a system of concurrently-running "piped" programs!

Available Now ... Only $50

Ask your Amiga dealer for ARexx and WShell, or order them directly from the author. Please include a check or money order plus $2/item shipping ($8 for overseas airmail) and any applicable taxes. Dealer inquiries welcome!

William S. Hawes
P.O. Box 308
Maynard, MA 01754
(508) 568-8695

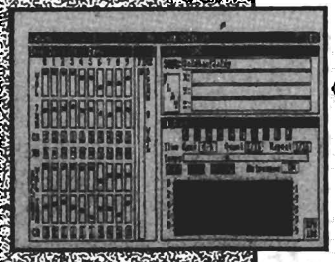
Amiga is a trademark of Commodore-Amiga, Inc.



Sound oasis

$99.95

The internal sound capabilities of the Amiga are better than that of any other personal computer. These capabilities mean nothing though, without quality digital sounds, which up till now have been scarce. Sound Oasis gives Amiga owners access to a large library of studio-tested digital samples, by using the Amiga's built in disk drive to read disks made for the Mirage Digital Sampling Keyboard. Sounds can then be played from a MIDI keyboard, the computer keyboard, or saved as an IFF standard file. Mirage is a trademark of Ensoniq Inc.



dynamic drums

$79.95

Transform your Amiga into a professional-quality drum machine with this software package. Easier to use than hardware-based drum machines because everything is displayed graphically on screen. Enter drum patterns quickly and easily in real time with visual feedback and editing. Create realistic drum tracks with any of the 100 drum and percussion samples that are included or use your own unique IFF one-shot samples. Dynamic Drums also has full MIDI implementation and even becomes velocity sensitive when triggered from a MIDI keyboard.

DYNAMIC STUDIO

$199.95

A powerful MIDI sequencer that takes full advantage of the Amiga's sound, graphics, and sophisticated user-interface. Dynamic Studio is perfect for professional applications due to its sophisticated editing capabilities and SMPTE support. It is also ideal for home studios, because in addition to sequencing MIDI instruments, Dynamic Studio has a built-in drum machine, and the ability to playback instruments translated with Sound Oasis.

No Copy Protection

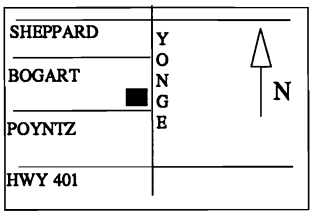
NEW WAVE SOFTWARE

P.O. Box 438 St. Clair Shores, MI 48080 (313) 771-4465

CANADA'S BEST SELECTION OF
AMIGA - C64 - C128 SOFTWARE
COMSPEC HARD DRIVES AND BERNOULLI DRIVES
AVAILABLE FOR
THE AMIGA 500 AND 2000

Electronics 2001
COMPUTER PRODUCT CENTRE

4736 YONGE STREET
WILLOWDALE ,ONT
M2N 5M6
TEL: (416) 229-2700



AUTHORIZED COMMODORE DEALER SINCE 1979

The Memory Location
COMMODORE
AMIGA™
Specialists
Hardware
Software
Service
Support
Peripherals
Accessories

Satisfaction Gauranteed!
An entire store dedicated to the *AMIGA*.
Hours: Mon.-Thur. 10-6 Fri. 10-8 Sat. 9-5
The Memory Location
396 Washington Street
Wellesley, MA 02181
(617) 237 6846
(617) 237 6440
We Ship UPS Mon.-Fri.

GET MORE SPIRIT . . . for your AMIGA™ A500 or A1000

world of commodore Booth #129

HARD DRIVE ADAPTOR
For ST—506 MFM or RLL Controllers and Drives.
• Use those inexpensive IBM type hard drives. • Enclosed in Amiga-matching metal chassis with passthru. • Includes installation disk with driver software. • Complete 20 and 40 MB systems available.
For A500/1000 **HDA-506...*Maximum Economy!***

2 MB INTERNAL MEMORY
Spirit's *IN*ternal expansion. *SIN*—and free your external bus.
• Expandable from 0K to 0.5, 1.0 and 2 MB. • Utilizes 1.0 Mb DRAMS. • Standard AmigaDOS auto-config with RAM on/off. • Optional external power supply. • Easy plug-in installation under FCC shield. • Populated boards available—best price on chips—call for QUOTE.
For A500 **SIN 500-2...*Absolutely The Best!***

MIDI STAR™ INTERFACE
Multiple Ports/Switches/Lights
• 2 IN, 6 OUT-OFF-THRU and RS-232 pass-through. • All ports switch selectable. • Provides distribution control and "star" configuration.
• Eliminates daisy chains and cable changes.
• Includes LED status and signal confirmation.
For all Amigas **MS-2x6...*Professional Value!***

1.5 MB INTERNAL MEMORY
Memory chip prices are coming down.
• Expandable from 0K to 0.5, 1.0 and 1.5 Mbytes.
• Includes battery-backed Clock/Calendar for A1000, optional for A500. • Internal design leaves expansion bus free. • Solderless installation under FCC shield.
• Install disk includes useful utilities. • Market Price populated boards available.
For A500/1000 ***...Super Low Pricing!***

Most major credit cards accepted...
ORDER TOLL FREE
(800) 433-7572

SPIRIT™ TECHNOLOGY

Spirit Technology Corporation
220 West 2950 South, Salt Lake City, Utah 84115
(801) 485-4233 • FAX (801) 485-6957
AMIGA is a trademark of Commodore-Amiga, Inc.
Dealer enquiries invited

BRiWALL

Our Third Year In Business!
We Know How To Service Our Customers!

SOLID PRODUCTS & SOLID SUPPORT

P.O. Box 129 / 58 Noble Street
Kutztown, PA 19530

24 HOURS — TOLL FREE
1-800-638-5757
OUTSIDE USA CALL 1-215-683-5433

SATISFACTION GUARANTEED
Earn Bonus $$
No Surcharge On Charge Orders
Friendly Service

ACCESSORIES

Control Center	$ 79
Cover Amiga CPU's	9
Cover Amiga Keyboard's	8
Cover Amiga 500	9
Cover Amiga Monitor	17
Flicker Master	14
Gender Changer/Parallel Adaptor	21
Lightpen for Amiga	89
Modem Cable-RS232 to A1000	22
Modem Cable for A500	22
Mousepad	8

BOOKS

Advanced Amiga BASIC Book	15
Amiga BASIC Inside & Out Book	20
Amiga DOS Express Book	20
Amiga DOS Inside & Out Book	17
Amiga DOS Reference Manual	19
Amiga Disk Drives Inside & Out	24
Amiga Hardware Manual	18
Amiga Machine Language Book	17
Amiga Programmer's Guide	15
Amiga System Programmer's Guide	27
Amiga Tips & Tricks Book	17
Amiga for Beginners	15
Beginner's Guide/Amiga	15
Elemenhary Amiga BASIC Book	13
Inside Amiga Graphics	15
Inside the Amiga With C	21
Intuition Manual	20
Kickstart Guide to Amiga	19
Machine Language-Amiga Book	17
ROM Kernal : Exec	18
ROM Kernal : Lib/Dev	24
Disks for Abacus Books	12

GENERAL BUSINESS

Acquisition	$189
Analyze 2.0	69
Becker Text	99
Citydesk v 2.0	129
Data Retrieve	52
Data Retrieve Professional	199
Excellence	169
Flow	65
Haicalc	39
KFS Accountant-Amiga	195
Kindwords	69
Logistix	69
Magellian	139
MaxiPlan Plus	125
MaxiPlan 500	95
Microfiche Filer	69
Microfiche Filer Plus	125
Microlawyer	39
Money Mentor 2.0	69
Organize	49
P.H.A.S.A.R.	69
Page Setter	90
Prowrite v 2.0	75
Professional Pages 1.1	249
Publisher Plus	64
Publishing Partner	129
Reason	249
Scribble	55
Security Analyst	65
Shakespeare	149
Softwood File II SG	69
Superbase PERSONAL	59
Superbase Personal 2	105
Superbase Professional	235
Textpro	59
Viziwrite	59
Word Perfect	199
Works (The)	125
Works - Platinum Edition*	175

EDUCATIONAL

Adventures of Sinbad	$ 32
Aesop's Fables	32
All About America	39
Decimal Dungeon	32
Discovery Math	27
Discovery Spell	27
First Letters/Words	30
First Shapes	30
Fraction Action	32
Kid Talk	30
Kinderama	32
Linkword: French	24
Linkword: German	24
Linkword: Italian	24
Linkword: Spanish	24
Math Talk	30
Math Wizard	32
Math-amation (College)	55
Mavis Beacon Typing	36
Perfect Score SAT	55
Read A Rama	32
Speller Bee	29
Surgeon	34
Word Master	32

IT IS OUR POLICY to stock what we advertise and carry the best products available. OVER 300 PRODUCTS in stock now and more arriving daily!! We carry all products by ABACUS, AEGIS, BROWN WAGH, DR. T, DISCOVERY, ELECTRONIC ARTS, FREE SPIRIT, GOLD DISK, MICRO ILLUSIONS, MINDSCAPE, NEW TEK, OXXI, PROGRESSIVE, PYSGNOSIS, UNICORN... and more. IF YOU DON'T SEE IT LISTED, PLEASE GIVE US A CALL.

ENTERTAINMENT

AAArgh	$ 25
Adventure Workshop	39
Alien Fires	25
Annals of Rome*	27
Arkanoid	20
Auto Duel	35
Balance of Power	36
Barbarian	28
Bard's II: Destiny Knight	39
Bard's Tale I	33
Battle Chess	39
Better Dead then Alien	22
Blackjack Academy	27
Breach	27
Bridge 5.0	25
Capone	27
Captain Blood	33
Carrier Command	32
Centerfold Squares	24
Chessmaster 2000	36
Clever & Smart	22
Corruption	32
Craps Academy	32
Defender of the Crown	33
Descartes	25
Destroyer	29
Double Dragon*	29
Dragon's Lair*	35
Earl Weaver Baseball	35
Ebonstar	27
Empire	36
F18 Interceptor	36
Faery Tales	32
Ferrari Formula One	36
Fire & Forget	29
Flight Simulator 2	35
GB Air Rally	25
Galileo 2.0	49
Gauntlet	36
Gone Fish'n*	35
Guild of Thieves	29
Gunshoot	22
Hardball Baseball	29
Hunt for Red October	36
Hybris	27
Ice Hockey-Super Star	35
Ice Hockey: Wayne Gretsky*	35
Jet	35
Kings Quest III	36
Knight Orc	29
Land of Legend	32
Leaderboard Dual Pak	26
Leaderboard World Class	29
Leisure Suit Larry	29
Lord of the Rising Sun	33
Main Frame	28
Marble Madness	36
Master Ninja*	29
Mean 18 Golf	27
Mobieus	39
Obliterator	28
POW	29
Phasar Gun	42
Port of Calls	32
Portal	27
Pub Games*	27
Questron 2	36
Reach for the Stars*	29
Rocket Ranger	33
Roger Rabbit	33
Romantic Encounters - Dome	27
Rubicon Alliance*	24
SDI	33
Scrabble	29
Sex Vixens from Space	29
Shadow Gate	33
Shangai	27
Sky Fox 2	29
Space Quest 2	36
Spacecutter	25
Star Fleet 1	39
Star Glider 2	32
Stellar Conflict	27
Strip Poker	29
Sword of Sodan*	34
TV Sports Football*	33
Test Drive	27
Thexder	25
Three Stooges	33
Tracers	27
Turbo	19
Twilight's Ransome	27
Ultima 3	33
Ultima IV	39
Universal Military Simulator*	35
Vampire's Empire	29
Virus	25
Vyper	22
Wizard Wars*	29
World Tour Golf	29
Zany Golf*	33
Zoom	20

CREATIVITY/GRAPHICS

Analytic Art	40
Animate 3D	99
Animation Effects	39
Animation Stand	39
Animator	85
Animator Apprentice	199
Bumper Sticker Maker	37
Business Card Maker	37
Calligrapher v 1.05	79
Celebrity Cookbook	27
Comic Setter	69
Deluxe Paint 2	86
Deluxe Photo Lab	99
Deluxe Print 2	55
Deluxe Productions	139
Deluxe Video 1.2	86
Digi-paint 3.0	40
Digiview 3.0	129
Director	49
Draw 2000 (formerly Draw Plus)	159
Dynamic Cad	299
Express Paint V3	69
Fantavision	49
Flipside	25
Graphics Studio	39
Home Builder's CAD	135
Intro CAD	52
Lights, Camera, Action	59
Page Flipper Plus FX	99
Photon Paint	65
Photon Paint Surface Disk	25
Photon Video Cel Animator	99
Photon Video Transport Controller	195
Pixmate	45
Printmaster Plus	33
Pro Video Plus	199
Sculpt 3D	69
TV Show	65
TV Text	65
Turbo Silver Plus-New Version	139
Video Scape 3D 2.0	125
Video Titler 1.1	99
X-Specs	92
Zuma Fonts-each	25

HARDWARE

Alegra Board-for A1000-2MB	$ 739
Alegra Board for A1000-0K	219
Alegra Board for A1000-512K	379
AmiGEN Genlock	169
Canon PJ1080A Color Ink Jet Prntr	519
Color Splitter	75
Copy Stand	59
Digidroid	69
Drive-22 Meg Hard-Phoenix	599
ECE Midi 1000	49
ECE Midi 500/2000	49
Easyl 1000 Draw Tablet	369
Easyl 2000 Draw Tablet	399
Easyl 500 Draw Tablet	329
Flicker Fixer/A2000	495
Frame Grabber	549
GV A2000 20MB Hard Card SCSI	619
GV A2000 2MB HardDisk Cntlr SCSI	309
GV A2000 40MB HardcardQuantam	839
GV A2000 45MB Hardcard SCSI	789
GV A2000 80MB HrdcrdQuantam	1259
GV A500 30MB HrdDrv/RAM SBsys	759
GV A500 45MB HrdDrv/RAM Sbsys	959
Gen/One Genlock	719
IVS Trumpcard SCSI Cntrller	139
Live! A1000 Framegrabber	275
Live! A500 Framegrabber	299
Modem-Supra 2400 w/o cable	139
Panasonic 16mm Lens NO iris	22
Panasonic 16mm Lens with iris	49
Panasonic W1410 Camera	189
Perfect Vision	169
Pro RAM for A2000	249
ProGEN Gen Lock	379
Starboard II OK Memory Board	269
Supergen	689
Xerox 4020 Color Ink Jet Printer	1149

SOUND/MUSIC

Deluxe Music Construction	$ 65
Dr T Midi Recording Studio	49
Dr. T KCS V1.6A	165
Dynamic Drums	52
Dynamic Studio	135
Instant Music	36
Midi Magic	99
Model A Midi Interface	49
Music Studio	34
Music-x	195
Perfect Sound A1000	65
Perfect Sound A500	65
Sonix	52
Sound Oasis	65
Studio Magic	65
Synthia	65

UTILITIES/LANGUAGES

64 Emulator 2/Amiga 500	$ 52
64 Emulator for Amiga 1000	52
A Talk Plus	52
AC/BASIC v 1.3	135
AC FORTRAN	199
Amiga Workbench 1.3	24
Assempro	64
Aztec C Developer Compiler	195
Aztec C Professional Compiler	129
Aztec C Source Level Debugger	49
BBS PC/New Version	99
Benchmark Modula 2	129
Butcher 2.0	25
C.A.P.E. 64K	59
CLI Mate	27
Cygnus Ed Prof Text Editor	69
DOS-2-DOS	39
DSM	49
Diga	52
Disk-2-Disk	35
Diskmaster	39
FACC II	25
GOMF V2.2	25
Grabbit	20
Inova Tools	52
Lattice C + +	339
Lattice C V 5.0 Developer Pack	205
Mac Emulator	call
On Line V2	42
Power Windows 2.5	59
Project D	35
Quarterback	45
True Basic 2.0	69
True Basic Runtime Pak	99
UltraDOS	39
Virus Infection Protection	33

Products marked with * were not released at time ad was prepared. Please call for firm price and availability.

Free Spirit Software Inc.

ULTRA DOS UTILITIES
MODULE I

Ultra DOS Utilities - Module I is the ultimate in file handling and backup system for the Amiga series of computers. Ultra DOS brings the simplicity and ease of use of the Intuition operating system to AmigaDOS to:

- Perform virtually all AmigaDOS functions at the click of a mouse button
- Copy/Delete files from/to any legal AmigaDOS device
- Copy by simple or complex wild card patterns
- Copy be creation date, volume device or directory
- Quickly and easily install even the most complex software on your hard drive
- Backup valuable files on a floppy or hard drive
- Perform whole hard drive backup with multi-format feature
- Preview & edit text files
- Preview IFF picture files
- Set & clear all types of file protection

only $39

CHRISTMAS CLASSICS

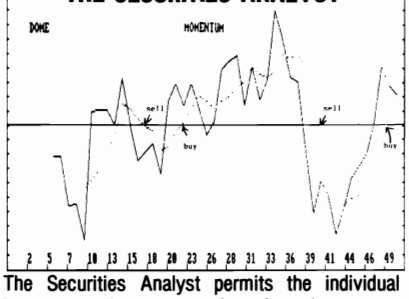
Enjoy your Christmas! Beautiful Christmas scenes are displayed while your computer plays your favorite holiday songs. Sing-along to Deck the Halls, Jingle Bells, Rudolph, Silent Night and more, as the lyrics to these Christmas classics scroll across the screen. Great for your holiday gatherings!

JUST $10

HOLIDAY SPECIAL!
Get your FREE
Christmas Classics
with any order over $100.
offer good until 12/31/88

only $19

THE SECURITIES ANALYST



The Securities Analyst permits the individual investor to chart any number of stocks over an extended period of time. Using this package you can make intelligent buy-sell decision based on analysis of market trends. Analysis charts may be displayed on the screen or output to printer. A comprehensive manual on Technical Analysis of Stock Market Trends is also included. Great Buy!

INTRO OFFER $65

BONUS DOLLARS

EARN BONUS $$$ WHEN YOU ORDER!!! For every $50 of software and accessories that you order, you earn one bonus dollar. That's an additional 2% discount!! Use your bonus dollars on future purchases.

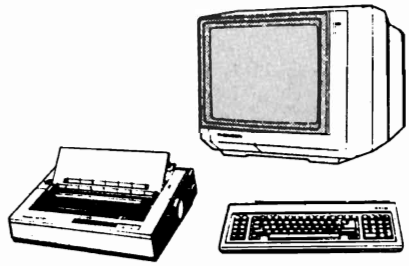
CATALOGS

We now have our ALL NEW AMIGA catalog, describing hundreds of products for your computer. Call or write for your copy of our AMIGA CATALOG today!!

BRiWALL
SOLID PRODUCTS & SOLID SUPPORT
P.O. Box 129 / 58 Noble Street
Kutztown, PA 19530
24 HOURS — TOLL FREE
1-800-638-5757

Circle 132 on Reader Service card.

HARDWARE



We carry a full-line of hardware including expansion boards, disk drives, monitors, printers and computers. Call for prices and shipping charges.

The Best Deal in Town! MAST presents the MATCHBOX Collection
Hi-Tech, Small Size, Low-Cost Peripherals for your AMIGA!!

Superslimline External Drives

UNIDRIVE
— $149

TWINDRIVE
— $259

880K External Floppy Disk Drives, just one and two inches high respectively. Compatible with ALL Amiga computers; Switch deselectable to disable drive(s); No Click; 3 Msec Step Time; Quality Fujitsu Drive; Special Circuitry which draws minimal power from your Amiga permits cool and reliable operation; Pass Thru on the Twindrive; complete with a Twelve Month Warranty!

Tiny Tiger™ — $229

A great cost-effective SCSI controller, not much bigger then the MAST Drives. Completely compatible with all Amigas; plugs into the Parallel port; Standard 25 pin Connector on rear for additional SCSI devices; Printer Port on back (yes will work at same time as your printer)! Call for hard drive prices!

Minimegs™ 500 — $165

For Amiga 500 owners, a breakthrough in memory design! In a tiny external case, you can fit up to 2 megabytes of RAM (in 512K increments) with full auto-configure plus a host of other features. A GREAT BARGAIN. Call for RAM prices.

As Captain Brad Stallion, you have been assigned a high priority mission by the Federated Government which has long been harried by a series of raids on the male population of its colonies. It is rumored that these raids are conducted by "The Tribe", a colony of beautiful female clones from the mysterious planet Mundo. Your mission is to locate the colony and destroy it! Please be advised that this is an ADULT adventure game!!

ONLY $29

ORDERING INSTRUCTIONS
For your convenience, you can place a phone order 24 HOURS/7 DAYS A WEEK on our TOLL-FREE LINE 1-800-638-5757. Price, Availability and Compatability Checks are also Welcome on our order line, Monday thru Friday, 9AM-4PM EST. AFTER HOURS, Orders Only Please!! When placing an order, please specify your COMPUTER MODEL, HOME & DAYTIME PHONE NUMBER, SHIPPING ADDRESS, METHOD OF PAYMENT and ITEMS ORDERED. To help us serve you better, please have all your information, including your CHARGE CARD number, ready before you call us.

ACCEPTED PAYMENT METHODS: We gladly accept payment by, PREPAID BY PERSONAL CHECK (will not hold for clearing) or MONEY ORDER: COD (continental USA only AND $4.00 ADDITIONAL); MASTERCARD; VISA; SCHOOL PURCHASE ORDERS. All payment must be in USA DOLLARS. THERE IS NO SURCHARGE FOR CHARGE CARDS and your card is NOT charged until we ship.

TECHNICAL SUPPORT
We do our very best to help you with your product selections, before you order and after you receive your product. General questions, price, and compatibility with your computer, etc. will be handled by our order staff at the time you place your order. BUT if you have specific, detailed questions about a product, printer compatibility questions etc., you will get the most help from our TECHNICAL SUPPORT LINE at 215-683-5699. Call Monday thru Friday, 9AM-4PM EST, and our trained tech staff will be happy to help you.

SHIPPING POLICY
ALL ORDERS received before 3PM EST will normally be shipped same or next business day. Out-of Stock items will be shipped by the same shipping method as original order, normally within 3 or 4 business days. All UPS shipments are sent SIGNATURE REQUIRED/NO DRIVER RELEASE. ADD the following shipping charges to your TOTAL software order.

UPS: GROUND-$4.00 (cont USA ONLY); AIR/RUSH-$7.00 (includes Puerto Rico/Hawaii); OVERNIGHT-$12.00 PLUS $2.50 per item (must be received by 12 NOON)

PRIORITY MAIL: USA-$4.00 (includes APO/FPO); CANADA/MEXICO-$6.00; OTHER FOREIGN-$4.00 handling PLUS Actual Shipping (minimum $12.00)

RETURN POLICY
We have a liberal return policy to better service your needs. Software piracy is a problem, but as long as our policy is not abused, we will continue to honor it. If within 15 days from the time you receive an item, you are not satisfied with it for any reason, you may return it to us for either a refund, exchange or open credit. REFUNDS ARE SUBJECT TO A 10% RESTOCKING FEE PER ITEM RETURNED ($5.00 MINIMUM PER ITEM). A DEFECTIVE ITEM will be replaced with the same item (NO CHARGE... of course). EXCHANGES/OPEN CREDITS will gladly be issued for the FULL PURCHASE PRICE OF THE ITEM.

ABBY'S DISCOUNT SOFTWARE

AMIGA SOFTWARE

A D &D: Hero of Lance	$ 29
4 x 4 Off Road Racing	$ 29
Cosmic Relief	$ 27
Carrier Command	$ 32
Tanglewood	$ 29
Questron II	$ 37
Fire & Forget	$ 29
Bubble Ghost	$ 25
Lights, Camera, Action!	$ 54
Fantavision	$ 44
Impossible Mission II	$ 35
Starglider II	$ 32
Ultra D.O.S.	$ 44
Pro Write 2.0	$ 92
Solitaire Royale	$ 22
Murder on the Atlantic	$ 29
Sub Battle Simulator	$ 37
Empire	$ 37
F/A 18 Interceptor	$ 37
Global Commander	$ 29
Bard's Tale II	$ 45
Deluxe Photo Lab	$109
Deluxe Print II	$ 60
Life or Death	$ 37
Space Quest II	$ 37
World Tour Golf	$ 29
Marble Madness	$ 37
Deluxe Paint II	$ 95
Marvis Beacon Teaches Typing	$ 37
Superstar Ice Hockey	$ 37
Zoom!	$ 22
Cracy Cars	$ 29
Arkanoid	$ 22
Capone	$ 29
Creature	$ 29
P.O.W.	$ 29
Phaser Gun	$ 39
Garrison II	$ 37
Jet	$ 37
Rocket Ranger	$ 37
Vampire's Empire	$ 32
Three Stooges	$ 37
Excellence	$229
Paladin	$ 29
Romantic Encounters	$ 29
Centerfold Squares	$ 22
Jinxter	$ 29
Alternate Realily: City	$ 29
Ebonstar	$ 29
Obliterator	$ 29
Ports of Call	$ 37
Disk 2 Disk	$ 30
Dos 2 Dos	$ 33
Kino Words	$ 60
Aaargh	$ 23
Photon Paint	$ 60
Diga	$ 59
Moebius	$ 44
Sonix	$ 59
On Line	$ 52
Faery Tail	$ 37
Fire Power	$ 18
Breach	$ 29
Barbarian	$ 29
Phanatasie III	$ 29
Digi View	$149
Flight Simulator II	$ 37
Dark Castle	$ 32
Print Master Plus	$ 37
Art Gallery 1, 2 or 3 ... ea.	$ 22
Auto Duel	$ 37
Def Con 5	$ 29
Video Titler	$ 99
Bridge 5.0	$ 25
Chessmaster 2000	$ 32
Contra	$ 29
Starfleet I	$ 39
Master Ninja	$ 29
Thexder	$ 25
Reach For The Stars	$ 29
Scrabble	$ 29
Zany Golf	$ 29
Gauntlet II	$ 37
Captain Blood	$ 37
Out Run	$ 37
G.F.L. Football	$ 32
Zork Trilogy	$ 37
Leather Goodesses of Phobos	$ 29
King of Chicago	$ 37
Defender of Crown	$ 37
Superbase Personal	$ 99
Boot Camp	$ 29
Director	$ 54
Dungeon Master	$ 29
Hunt For Red October	$ 37
Instant Music	$ 37
Intellitype	$ 37
Leatherneck	$ 29
Major Motion	$ 29
Money Mentor	$ 79
Operation Wolf	$ 29
Police Quest	$ 37
Tobruk	$ 29
Twilight's Ransom	$ 27
Ultima IV	$ 42
U.M.S.	$ 37
Virus Infection Protector	$ 29
Western Games	$ 27

C64/128 SOFTWARE

Advanced Dungeons & Dragons:	
Pool of Radiance	$27
Heroes of the Lance	$27
Dungeon Masters Assistant	$27
Zak McKracken	$25
Sporting News Baseball	$29
Boot Camp	$22
Napoleon in Russia	$22
Maniac Manor	$25
Wizardry:	
Knight of Diamonds	$29
Proving Grounds	$29
Platoon	$22
Home Video Producer	$37
Metrocross	$18
Mainframe	$25
Murder on the Atlantic	$25
Ultima V	CALL
Bard's Tale III	$29
Karnou	$22
Clubhouse Sports	$22
Rockford	$22
Award Maker Plus	$29
Betterworking Word Publisher	$29
Crossbow	$22
Alf	$12
Road Runner	$25
Road Wars	$22
Impossible Mission II	$29
Wasteland	$29
Contra	$22
G.E.O.S. 64 or 128	CALL
Print Shop	$32
P.S. Graphic Library 1, 2 or 3 ... ea.	$18
P.S. Companion	$25
P.S. Holiday Edition Graphic Library	$18
Carmen San Diego (U.S.A.)	$29
Carmen San Diego (World)	$25
Carmen San Diego (Europe)	$29
Leather Goddesses	$25
Zork Trilogy	$32
Ogre	$22
Family Feud	$10
Jeopardy	$10
Wheel of Fortune	$10
Dark Castle	$25
Sky Travel	$34
Blackjack Academy	$29
Renegade	$24
Arkanoid	$20
Alcon	$20
Bubble Bobble	$24
Rastan	$24
Skate or Die	$22
Paper Boy	$25
Marble Maddness	$22
Gunship	$25
L.A. Crackdown	$29
4th & Inches	$22
Certificate Maker	$29
Newsroom	$14
Echelon	$32
Three Stooges	$27
Battle Droidz	$18
Red Storm Rising	$29
Global Commander	$22
Aliens Fires	$22
4 x 4 Off Road Racing	$27
1541/1571 Disk Alignment	$22
Legacy of the Ancients	$22
Strike Fleet	$22
Sargon III	$14
Flight Simulator	$37
Flexidraw 5.5	$25
Stealth Fighter	$29
Term Paper Writer	$29
California Games	$29
Hard Ball	$22
Auto Duel	$36
L.B. Golf Triple Pack	$14
Hunt for Red October	$29
Power At Sea	$22
General Accounting System 64/128	CALL
Big Blue Reader 64/128	$35
Warpspeed	$37
Paperclip III	$37
Paperclip Publisher	$37
Monopoly	$22
Scruples	$29
Test Driver	$22
Echelon	$32
Chuck Yeager's A.F.S.	$25
Might & Magic	$29
Copy II	$25
Captain Blood	$25
Alien Syndrome	$22
Out Run	$27
Road Runner	$27
Space Harrier	$27
Caveman Ugh-Lympics	$22
Double Dragon	$25
Starfleet I	$29
Chessmaster 2000	$29
The Eternal Dagger	$29
Sons of Liberty	$25
American Civil War	$29
American Civil War Vol. 1 or 2 ... ea.	$29
High Rollers	$12
Contra	$22

DUST COVERS

Amiga 2000 w/Keyboard	$15.00	Vic 20/C64 Computer	$ 7.00
Amiga 1000 Computer	$ 9.00	C-128 Computer	$ 7.00
Amiga 500 Computer	$ 9.00	C-128D w/Keyboard	$15.00
Amiga Keyboard	$ 7.00	C-128D Keyboard	$ 7.00
1010 Disk Drive	$ 7.00	1541 Disk Drive	$ 7.00
1020 Disk Drive	$ 7.00	1571 Disk Drive	$ 7.00

RIBBONS

Commodore 1525	$7.50	Star NX/NP/NL-10	ea. $8.00
Commodore 1526/MPS 802	$7.50	Panasonic 1080i/91i/92i	$7.95
Commodore MPS 801	$7.50	Okimate 10/20 (Black)	$4.50
Commodore MPS 803	$7.00	Okimate 10/20 (Color)	$5.00
Commodore MPS 1200	$7.00	Star 1000/1000C (Black)	$6.00
Star Gemini (Black)	$2.50	Star 1000/1000C Rainbow	$9.00

ETC.

Ergo Stick Joystick	$19.95	Mouse Master	$35.95
Epyx 500XJ Joystick	$15.00	C-64 Power Supply	$29.00
Vic 1351 Mouse	$35.00	C-128 Power Supply	$50.00
Mouse Mat	$ 7.50	Amiga 500 Power Supply	$79.00
Mouse House	$ 4.95	Super Graphic Interface	$55.95
Mouse Holder	$ 4.50	Super Graphic Jr. Interface	$35.95

Order Line
1-800-282-0333

10 a.m.-8 p.m. Monday-Thursday
10 a.m.-6 p.m. Friday
10 a.m. to 3 p.m. Saturday

Customer Service & Ohio Residents
1-513-879-9699

11 S. Wright Avenue, Fairborn, OH 45324

SOFTWARE ONLY - Add $5 for shipping. Hardware and all orders requiring shipment via US Post Office are subject to additional freight charges. Charge card orders add $2 service charge. No waiting when paid by credit card, certified cheque or money order. All items subject to availability and price change. PLEASE CITE AD NUMBER WHEN ORDERING. ALLRETURNS MUST BE ACCOMPANIED BY AN AUTHORIZATION.

AD #CAN

Reader Service Card

Fill out this card (please print) or attach your peel-off mailing label, and circle the reader service number for the ads that you'd like more information about (see Advertiser Index below).

Name _____

Company _____

Address _____

_____

Postal/Zip Code _____

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30

TA V1I4

Advertiser	Page	Reader Service #
A.S.D.G.	61	1
Abby's Discount Software	80	2
Briwall	78-79	3
Comspec Communications Inc.	Outside Back Cover	4
Digital Depot	24	5
Electronics 2001	77	6
Emerald Intelligence	Inside Front Cover	7
Gold Disk Inc.	24	8
Inovatronics	1	9
Intercomputing	25	10
Interface Technologies	7	11
MCP	11	12
New Horizons	Inside Back Cover	13
New Wave Software	76	14
PC Plus	61	15
Spirit Technology	77	16
The Memory Location	77	17
UEdit	24	18
William S. Hawes (ARexx)	76	19

Advertising in Transactor for the Amiga

Interested in advertiising in *TransAmi?* Call Max Jacobson at 416-764-5274 NOW!

for the AMIGA Transactor

The Magazine for Amiga Programmers

Please print your name and address below (or transfer your mailing label)
Check here for change of address ☐

Name: _____

Address: _____

City: _____ *Province/State:* _____

Zip/Postal: _____ *Phone:* _____ *CompuServe ID:* _____

NOTE: Prepayment required. Purchase orders will be accepted ONLY if accompanied by payment.

Subscribers only: please enter subscription number from mailing label ☐

Subscription Order Section

	Canada ($Cdn.)			U.S.A. ($US)			Foreign ($US)			Air Mail ($US, overseas only)		
	Mag	Disk*	Both*	Mag	Disk	Both	Mag	Disk	Both	Mag	Disk	Both
1 year	☐19.00	☐ 67.00	☐ 86.00	☐15.00	☐ 55.00	☐ 70.00	☐21.00	☐ 65.00	☐ 86.00	☐ 40.00	☐ 79.00	☐119.00
2 years	☐35.00	☐123.00	☐158.00	☐27.50	☐101.00	☐128.50	☐38.50	☐119.50	☐157.50	☐ 73.00	☐145.00	☐218.00
3 years	☐47.50	☐167.50	☐215.00	☐37.50	☐137.50	☐175.00	☐52.50	☐162.50	☐215.00	☐ 100.00	☐197.50	☐297.50

*** Ontario residents please add 8% of disk subscription price for provincial sales tax. No PST on magazine subscriptions.**

☐ **Please send Microfiche instead of magazines** ☐ **Send me info on Transactor Classic (C64/128)**

Combo Subscription Bonus!

If you are subscribing to BOTH the magazine and the disk, check one of the following FREE gifts:

☐ **Transactor Classic Amiga Disk #1** ☐ **Fish Catalog Disk** ☐ **Extended Subscription (7 Issues)**

Product Order Section

Ontario residents please add 8% PST on all items

$Cdn.	$U.S.	
☐ 14.95	12.95	Transactor Classic Amiga Disk #1
☐ 14.95	12.95	Transactor Classic Amiga Disk #2

Fish Disks $6.95 US, $7.95 Cdn. each

Note: Disk numbers not yet released may be ordered in advance - simply add numbers to end of list. Disks 57, 80 and 88 are not available.

☐ 1	☐ 11	☐ 21	☐ 31	☐ 41	☐ 51	☐ 62	☐ 72	☐ 83	☐ 94	☐ 104	☐ 114	☐ 124	☐ 135
☐ 2	☐ 12	☐ 22	☐ 32	☐ 42	☐ 52	☐ 63	☐ 73	☐ 84	☐ 95	☐ 105	☐ 115	☐ 125	☐ 136
☐ 3	☐ 13	☐ 23	☐ 33	☐ 43	☐ 53	☐ 64	☐ 74	☐ 85	☐ 96	☐ 106	☐ 116	☐ 126	☐ 137
☐ 4	☐ 14	☐ 24	☐ 34	☐ 44	☐ 54	☐ 65	☐ 75	☐ 86	☐ 97	☐ 107	☐ 117	☐ 127	☐ 138
☐ 5	☐ 15	☐ 25	☐ 35	☐ 45	☐ 55	☐ 66	☐ 76	☐ 87	☐ 98	☐ 108	☐ 118	☐ 128	☐ 139
☐ 6	☐ 16	☐ 26	☐ 36	☐ 46	☐ 56	☐ 67	☐ 77	☐ 89	☐ 99	☐ 109	☐ 119	☐ 129	☐ 140
☐ 7	☐ 17	☐ 27	☐ 37	☐ 47	☐ 58	☐ 68	☐ 78	☐ 90	☐ 100	☐ 110	☐ 120	☐ 130	☐ 141
☐ 8	☐ 18	☐ 28	☐ 38	☐ 48	☐ 59	☐ 69	☐ 79	☐ 91	☐ 101	☐ 111	☐ 121	☐ 131	☐ 142
☐ 9	☐ 19	☐ 29	☐ 39	☐ 49	☐ 60	☐ 70	☐ 81	☐ 92	☐ 102	☐ 112	☐ 122	☐ 133	☐ 143
☐ 10	☐ 20	☐ 30	☐ 40	☐ 50	☐ 61	☐ 71	☐ 82	☐ 93	☐ 103	☐ 113	☐ 123	☐ 134	☐ 144

The Transactor for the *AMIGA* Disk $12.95 US, $14.95 Cdn. each
(these disks correspond to issues of **Transactor for the *AMIGA***)

☐ Vol. 1, Iss. 1 ☐ Vol. 1, Iss. 2 ☐ Vol. 1, Iss. 3

Add flat $2.00 shipping & handling for any number of items ordered from above box.

Transactor for the *AMIGA* Back Issues and Microfiche: $4.50 U.S., $4.50 Cdn. (MF = Microfiche)

☐ Vol. 1, Iss. 1 ☐ MF ☐ Vol. 1, Iss. 2 ☐ MF ☐ Vol. 1, Iss. 3 ☐ MF

Ontario residents please add 8% provincial sales tax for any items ordered from this box.

☐ Visa ☐ MasterCard ☐ Cheque/MO enclosed Cheque #_____ Dated ___/___/___ Amount _____

Acct. # _____ Expires ___/___ Signature _____

Please allow 6-8 weeks for delivery in the U.S., 4-6 weeks in Canada.

FOLD ALONG SCORES WITH CORRECT REPLY CARD ON OUTSIDE. USE TAPE, PLEASE - NO STAPLES OR GLUE.

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 390 BUFFALO, NY
POSTAGE WILL BE PAID BY ADDRESSEE

Transactor
P.O. Box 338 Station 'C'
Buffalo, NY, 14209-9990

place stamp here

Transactor
10-85 West Wilmot Street
Richmond Hill, Ontario, Canada
L4B 1K7

Business Reply Mail
No Postage Stamp Necessary
If mailed in Canada

Postage will be paid by:

Canada Post Postes Canada 46487

Transactor
501 Alden Road
P.O. Box 3250
Markham Industrial Park
Markham, Ontario
L3R 9Z9

Introducing our new...

TOLL-FREE ORDER LINE
1-800-248-2719 Extension 911

Here's a convenient way to order magazine or disk subscriptions and other products. Nothing to fill out or drop in the mail! Just call the toll-free number above and have your credit card number ready. Available in the United States only.

NOTE: The toll-free number can be used for ordering only; for subscription queries, etc., call our office directly.

Color Graphics.

Multiple Fonts.

Spelling Checker.

Mailmerge.

Multiple Windows.

Resizable Graphics.

Fast Printing.

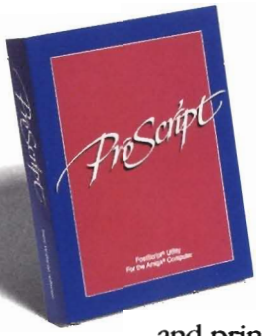
WYSIWYG.

No Other Word Processor For Amiga® Stacks Up To ProWrite™ 2.0.

Word is out. ProWrite 2.0 allows you to expand your Amiga word processing to new horizons. Use it to generate proposals with impressive bar charts. Create letters and reports with snappy graphics. Generate television storyboards. Even dash out newsletters with spot illustrations.

And of course, ProWrite 2.0 has all the other features you expect: a spelling checker with 95,000 word dictionary. Mailmerge. And fast graphics printing. Best of all, it works with our font packages, ProFonts I and II, giving you a broad range of professional and decorative fonts.

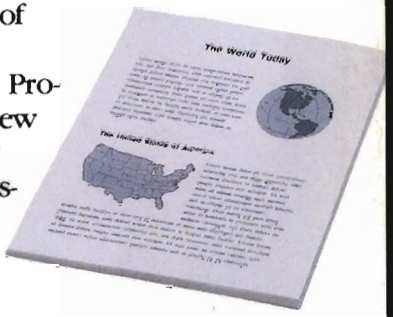
All this, plus ProWrite's powerful editing capabilities and ease-of-use combine to make ProWrite the best word processor for the Amiga computer.

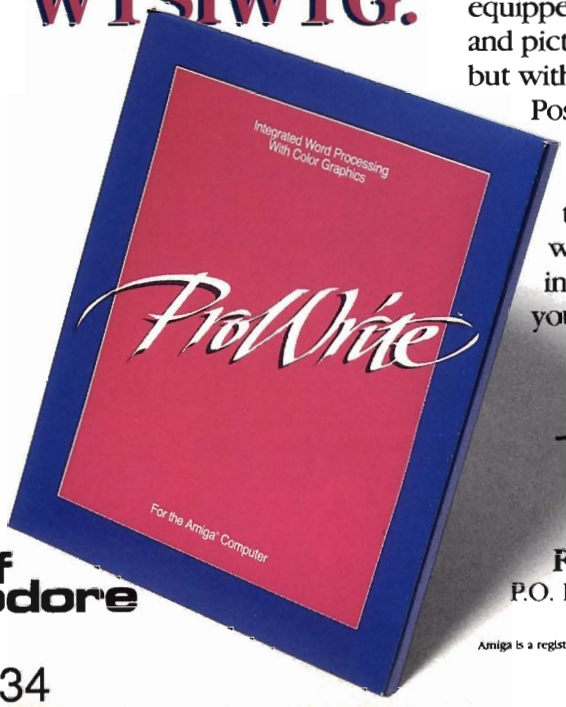


ProScript: Perfection In Print For The Amiga.

ProScript is the new PostScript utility from New Horizons Software that gives you the professional look of typeset copy. ProScript reads ProWrite files and prints them on any PostScript equipped printer, giving you the same multiple font and picture capabilities you expect from ProWrite, but with the typeset quality of PostScript printing.

ProWrite 2.0 and ProScript. A powerful new team that puts a new world of word processing and printing at your fingertips.





New Horizons
S O F T W A R E

First in Personal Productivity and Creativity
P.O. Box 43167/Austin, Texas 78745/(512) 328-6650

ProWrite, ProFonts and ProScript are trademarks of New Horizons Software, Inc.
Amiga is a registered trademark of Commodore-Amiga, Inc. PostScript is a registered trademark of Adobe Systems, Inc.

world of commodore

BOOTH #234

COMSPEC'S SEVEN BITS TO SPICE UP YOUR BYTES

MCS 1000
Connect any IEEE type printer to virtually any computer utilizing standard Centronics output. Plugs into a parallel port on your computer.

AX 2000
1 or 2 MB FAST RAM for the Amiga 1000 computer. Provides the extra RAM necessary to run advanced applications quickly and efficiently. Runs with no wait states. Complete with non-volatile RAM disk.

SD-20 SD-40 SD-60
Auto-booting external hard drives for the Amiga. The Comspec Hard Drive system consists of a SCSI Host Adaptor * and a Hard Drive Chassis, available in 20, 40, and 60 MB sizes. The only Hard Drive system for the Amiga that provides auto-booting under WB 1.2.

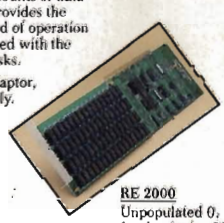
* Also available separately.

BN 20
Bernoulli 20MB 5 1/4" removable cartridges allow you to store sensitive data in a secure place. Also allows for the sharing of large amounts of data between computers. Provides the storage space and speed of operation of a hard drive combined with the portability of floppy disks.

Requires SCSI host adaptor, also available separately.

MCS 8000
Multi-user printer network/buffer which allows one printer to do the work of eight. Fully compatible, the MCS 8000 allows up to eight computers to share one printer. A must for any multi computer environment.

MCS 6550
256K print buffer. Frees up computer time by allowing you to process and print at the same time. Works alone or can be combined with the MCS 8000 to increase memory. No installation is required.



RE 2000
Unpopulated 0, 1 or 2 MB FAST RAM for the Amiga 2000. Similar to the AX2000 with 1/6 power consumption of competitive products. This highly reliable memory board is compatible with all standard Amiga hardware and software.

Available from:

COMSPEC

74 Wingold Ave., Toronto, Ont., M6B 1P5 (416) 785-3553 FAX 785-3668

Distributed worldwide by:

Precision Software Ltd.
6 Park Terrace
Worchester Park
Surrey KT4 7J2
England
01 330-7166
Fax 01 330-2089

Microton
Bahnofstrasse 2.
Postfach. 40 Ch 2542
Pieterlen, Switzerland
032 872-429
Fax 032 872482

Data Stream
29 Caryle St.
Naiper, New Zealand

Run Informatique
62 Rue Gerrard
Paris, France
45 81 5144

Airstat of Canada Inc.
3575 — 14th Ave.
Markham, Ont. Canada
L3R 2L6
1-800-263-3314
Fax (416) 470-0805

Interest A.S.
Markveien 44B
P.O. Box 59
4030 Hinna, Norway
(04) 881639
Fax (04) 881969

*Specifications subject to change without notice